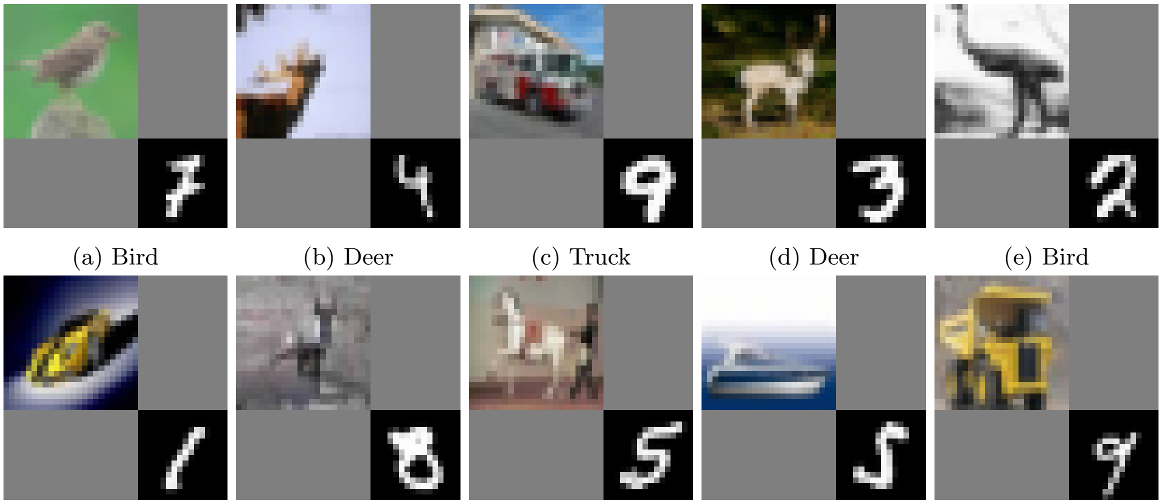
Project 2: Convolutional Neural Networks

CSE 849 Deep Learning (Spring 2025)

Gautam Sreekumar
Instructor: Zijun Cui

February 27, 2025

1 Image Classification using a CNN



(a) Bird (b) Deer (c) Truck (d) Deer (e) Bird

(f) Automobile (g) Deer (h) Horse (i) Ship (j) Truck

Figure 1: Sample images from the image classification dataset used in this project. The true class of each image is shown in its caption, with labels ranging from 0 to 9.

In this assignment, you will implement a convolutional neural network (CNN) to classify some *composite* images constructed from *sub-images* taken from CIFAR-10 and MNIST datasets into 10 classes. See fig. 1. The sub-image at the top left of each composite image is taken from the CIFAR-10 dataset and the label of the composite image (that we wish to predict) is always the same as the label of this CIFAR-10 image. In the bottom right of each composite image, there is a sub-image from the MNIST dataset and its label is not of interest to us. The label of the MNIST sub-image may or may not match the CIFAR-10 sub-image's label (which is also the ground truth label for the composite image).

In about 64% of the training samples, the numerical label assigned to the CIFAR-10 sub-image (the ground truth label for the composite image) coincides with the numerical label assigned to the MNIST sub-image, even though they represent different types of objects (e.g., a class ”3” in CIFAR-10 might be a cat, while the class ”3” in MNIST is the digit 3). As a result, your model may exploit this correlation and “cheat” by using features from the MNIST sub-image to predict the label of the composite image, instead of focusing on the relevant CIFAR-10 sub-image[1]. If your model cheats, it will perform poorly on validation and test sets due to the absence of correlation between MNIST and CIFAR-10 labels in the validation and test sets. Therefore, the challenge of this project is to learn a

[1] The reason why we suspect the model to cheat using MNIST features is because MNIST images are easier to classify than CIFAR-10 images.

robust model that accurately predicts the labels of composite images without relying on any shortcut, despite the bias in the training set – where CIFAR-10 and MNIST sub-images often share the same numerical labels.

In the second part of your assignment, you will visualize the filters of this CNN to know *what* your model used to make predictions.

1.1 Setting up the data

Downloading the dataset: You can download the dataset zip file from here. Use your MSU Google Drive to access it. The dataset link should take you to a zip file. The zip file can be extracted to a folder with three subfolders – `train`, `val`, and `test`. `train` and `val` will, in turn, contain 10 folders with each folder containing the images belonging to the same class. `test` folder will contain 10,000 images on which you will run your trained model. The predicted class for the test images must be submitted as a text file, like how you did in your previous assignments.

Creating the dataset class: To load the image classification dataset, you can use Torchvision's ImageFolder dataset. This will help you load `train` and `val` sets. But you will need to create your own dataset class for the `test` set since it doesn't follow the folder structure as `train` and `val`. Hence, your `ImageFolder` class will not work for `test` set. Alternatively, you can also run your model on the test images one by one but *make sure you pass it through the appropriate image transformations (same as what was used for validation dataset) before passing it to the model.*

Image preprocessing: The images must be processed before being fed into the model. A typical image preprocessing pipeline for an image classification model consists of the following: (1) randomly horizontal flip (only during training), (2) convert PIL image to tensor, and (3) channel-wise normalization. Check out this example. To normalize the images, use $[0.438, 0.435, 0.422]$ and $[0.228, 0.225, 0.231]$ as channel-wise mean and standard deviation respectively. The transformations can be composed and passed as an argument to the `ImageFolder` class which will apply these transformations to each image.

Along with these preprocessing steps, you can also include data augmentation steps to avoid overfitting. These techniques might be particularly useful since you are aware of the misleading correlation between the sub-image labels in the training set. We suggest GaussianNoise and RandomErasing as two useful data augmentation classes for this assignment. See this page for the list of available data augmentation classes.

Note: including too many data augmentations in the preprocessing pipeline can increase your training time, and some augmentations may even be detrimental to your model's performance.

Creating the dataset loaders: After creating the dataset class, create their corresponding dataloaders similar to the previous assignments.

1.2 CNN Architecture

You will implement a 5-layer CNN that consists of 2d convolution, batch normalization, ReLU, max-pooling, average pooling, and linear layers as shown in fig. 2.

Convolutional layers: The first convolutional layer will have 3 input channels since your image will have 3 channels (RGB). The number of output channels in each convolutional layers is mentioned in each box in fig. 2. The values for the arguments in the convolutional layers are given in table 1. All convolutional and pooling layers will use a stride of 1.

Batch normalization (BN): The BN layer inputs the number of input channels as its argument. This will be the same as the number of output channels of the previous convolutional layer.

Max pooling: Max pooling reduces the spatial dimensions using a local voting mechanism that leaves only the larger elements along the spatial dimensions. It has no parameters and therefore can be reused in the implementation.

Average pooling: The average pooling layer after the final convolutional layer will average the model's *features* over the height and width dimensions. Although you may use AvgPool2d for this assignment, it is easier to use AdaptiveAvgPool2d since it saves you the trouble of having to manually compute the size of averaging kernel. Read the linked documentation to learn how to use it.

Linear layer: The linear layer, generally referred to as the "fully connected" layer in various implementations, will map the output from ~~128~~ dimensions to 10 dimensions since we are solving a 10-way classification problem. In the end, the shape of your model's output will be batch size × 10. The final output of the model is called the *logits*. At the time of inference, the logit with the highest value will correspond to the predicted class. For example, if the output logits (ignoring the batch size) of a 3-way classification model are $\begin{bmatrix}0.3\\2\\1\end{bmatrix}$, then the predicted class is the second element with the value 2. This is equivalent to choosing from the class probabilities calculated using softmax.

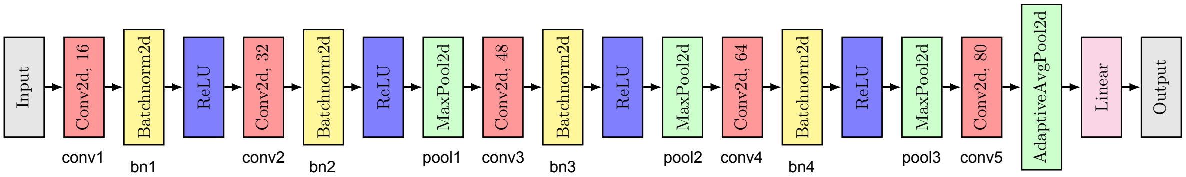


Figure 2: CNN architecture to be implemented in this assignment

Layer index	Kernel size	Padding	In channels	Out channels
Conv 1	7	3	3	16
Conv 2	3	1	16	32
Conv 3	3	1	32	48
Conv 4	3	1	48	64
Conv 5	3	1	64	80

Table 1: Parameters for each convolutional layer. The index corresponds to their position from the left of fig. 2.

1.3 Training the model

Since this is a classification problem, you will use the cross-entropy loss. Check the linked page to learn how to use it for an image classification problem. Read "hyperparameter tuning" in the Section 3. Important Tips on how to tune your model.

2 Analyzing your trained CNN

Congratulations! Your CNN model is trained and can now classify images fairly accurately. But *why* does it work? How does it obtain meaningful classifications from the information represented as image pixels? How does it know why some combinations of pixels form an airplane while others form a car? This is a difficult question to answer since your CNN contains several parameters that were automatically tuned by minimizing a classification error. You had little control over what your model learned. In this section, we will rely on some qualitative techniques to analyze your trained model.

2.1 Visualizing the filters

Each convolutional layer comprises filters that are parametrized by their weights and biases. The weights of a filter can tell us what that filter is looking out for. However, it is not always possible to directly visualize the weights of a filter since the input space of a filter could be high-dimensional.

For example, the fourth convolutional layer acts on a 48-dimensional space. Therefore, visualizing the filter weights directly is only suitable for understanding the first convolutional layer. Empirical evidence [ZF14; KSH12] suggests that the initial layers of a CNN capture low-level features such as edges and corners. You can verify this claim yourself.

In this assignment, we will visualize individual kernel filters from the first layer of your trained CNN. Remember that there are 16 output channels in the first layer, which means there are 16 kernel filters that map from the 3-channel RGB image space. Read the weight attribute from the first convolutional layer. It will be of the shape out channels $\times$ in channels $\times$ $k \times k$ where k is the kernel size. Use `imshow` function from Matplotlib to visualize **all** the filters from the first convolutional layer. Read the documentation of `nn.Conv2d` to know more.

2.2 Comparing the early and the later layers of your model

As we explained, visualizing the filter weights is unsuitable for analyzing filters that act on high-dimensional spaces. Therefore, we will take an alternative approach to analyzing the later layers.

Suppose your input image I is of the shape $3 \times h_{\text{in}} \times w_{\text{in}}$. After this image passes through all the layers up to (including) the fourth convolutional layer, you get your features F with dimensions $64 \times h_{\text{out}} \times w_{\text{out}}$, where $h_{\text{out}} < h_{\text{in}}$ and $w_{\text{out}} < w_{\text{in}}$. For ease, we will denote F as $[F_1, \ldots, F_{64}]$ where each F_i is a $h_{\text{out}} \times w_{\text{out}}$ grid. F_i's are the outputs of different functions acting on the input image. That is, $F_1 = g_1(I)$, $F_2 = g_2(I)$, and so on. $g_1, \ldots, g_{64}$ are functional compositions of all the layers up to (including) the fourth convolutional layer, and differ from each other only on which filter in the fourth convolutional layer is used. g_1 uses filter 1 in the fourth convolutional layer, g_2 uses filter 2, and so on. Therefore, comparing the outputs of $g_1, \ldots, g_{64}$ can tell us what the differences between filters in the fourth convolutional layer are. But, how do we compare the outputs of these functions?

If the input image contains a concept that is of interest to g_1, then F_1 will contain elements with larger absolute values. Thus, we can measure how much and how frequently the i^{th} filter in the fourth convolutional layer is activated by the images in our dataset by calculating the norm of F_i. Furthermore, we will compute the classwise average activation of a filter to check if any filter is consistently activated by a given class. If so, it means that the studied filter's purpose in the model is to look out for image patches that are representative of that class.

Here are the **step-by-step instructions** to compute the activations:

1. Iterate over the image-label pairs from your validation dataloader.

2. Get the intermediate features from your CNN. There is a provision in your starter code to achieve this. Set `intermediate_outputs = True`.

3. Compute the channelwise norm of the features after your first convolutional layer and the fifth convolutional layer. If your feature is of shape $b \times c \times h \times w$, your computed norm will be of shape $b \times c$. See `torch.norm` on how to compute the norm of features, particularly the `dim` argument.

4. Maintain the classwise norm values for every filter. You should keep track of $10 \times c$ values for both the first and the last convolutional layers. The starter code already has some provisions for this.

5. Plot the average classwise norm for each filter. Use the bar plot function from Matplotlib. In the end, you will have $16 + 80 = 96$ bar plots.

6. Analyze the plots corresponding to the filters from the first and the last layers. Include a descriptive analysis in your report. **Hint**: You can look into the variance across the classes for each filter. Are they uniform? What do activations that are uniform across the classes mean?

Most of the code to plot the activations is included in the starter code since the objective of this course is not to learn plotting using Python. Please reach out to the TA with any questions regarding the use of Python libraries.

3 Important tips

- Remember to not use any random image transformations during testing or validation as it can affect the reliability of your results.

- Your CNN will be a Python class with an `__init__` function containing the list of layers and a `forward` function handling forward computation. Make sure this class inherits from the `nn.Module`.

- Since the ReLU activation function does not have any parameters, you can reuse a single activation function module. You can also use a functional activation function.

- The output of the final pooling will have the shape batch size × ~~128~~ × 1 × 1. Therefore, you will need to "squeeze" out the last two unnecessary dimensions before passing it to the linear layer.

- Similar to your previous assignments, remember to set the model to train and eval modes before training and testing respectively. Also, zero out the gradients using the optimizer before the backpropagating through the loss.

- Set the clip argument to False in GaussianNoise if you are using it after normalizing transformation.

- **Hyperparameter tuning:** Generally, you only need minimal hyperparameter tuning if you are solving a "standard" problem using a suitable architecture (meaning your expected solution makes sense, not self-contradictory, etc.). But hyperparameter tuning is still useful to utilize your final training budget smartly. The models for this assignment can be trained within 40 epochs with sufficiently well-tuned hyperparameters. So in this assignment, you can **optionally** play around with learning rate, epochs, batch size, and learning rate scheduler to train your model quickly. We will provide a quick primer on the `CosineAnnealingLR` scheduler that is popular for training image classification models. You may also try other learning rate schedulers.

  `CosineAnnealingLR` scheduler slowly decays the learning rate from its initial value $\eta_{\max}$ that you set in the optimizer to a specified minimum value $\eta_{\min}$ over a specified number of steps $T_{\max}$ following a cosine curve. The steps are counted every time you call the `step` method of the scheduler (which you generally do at the end of an epoch). Some examples of this scheduler are shown in fig. 3. The general principles are that (1) there is more to learn during the initial stages of training (hence, a higher learning rate), and (2) a small training step size will slowly bring your model to the optimum towards the end.

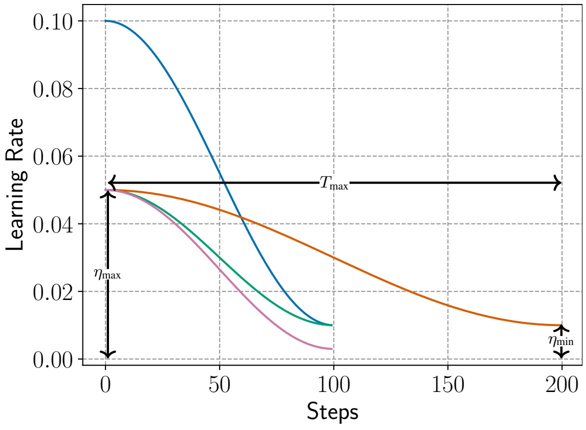


Figure 3: How various arguments in the learning rate scheduler affects the learning rate decay.

4 Submission Deliverables and Formats

4.1 Deliverables

- Question 1: Image Classification

  1. Completed Python files `q1.py`, `data.py`, and `model.py`.
  2. Saved model checkpoint
  3. Report the validation accuracy in your report (only PDF files)
  4. Predictions on the test set as a text file (see "formats" below)

- Question 2: Visualizing the filters

  1. Completed Python files `q2.py` and `q3.py`.
  2. Visualize all the filters from the first layer of your best model. Do these filters remind you of any patterns related to any of these CIFAR-10 classes?
  3. Include a few bar plots corresponding to filters from the first and the final convolutional layers.
  4. Explain what difference you note between the bar plots from these two layers. How can you use them to explain what your model has learned?

4.2 Formats

Submit a folder named `first name_last name_project_2` using the same subfolders as below:

1. code

   (a) q1.py

   (b) data.py

   (c) model.py

   (d) q2.py

   (e) q3.py

2. results

   (a) report.pdf

   (b) q1_model.pt

   (c) q1_test.txt

5 Grading Policy

1. In total, 100 points.

2. 60 points for submission completeness and functionality. q1.py (10 pts), q2.py (10 pts), q3.py (10 pts), data.py (10 pts), model.py (10 pts), report (10.pt). Your output is automatically graded using a Python evaluation script. Therefore, make sure to follow the details in the question accurately.

3. 20 points for the accuracy of your prediction on test set. You will be graded on the following rubric on your test accuracy:

   - $\geq 75\%$: 20 pts
   - $\geq 70\%$, but $< 75\%$: 15 pts
   - $\geq 65\%$, but $< 70\%$: 10 pts
   - $\geq 60\%$, but $< 65\%$: 5 pts

   - < 60%: 0 pts

   Make sure to submit your test-set predictions as `q1_test.txt` from your best-performing model chosen based on its validation accuracy. Also submit the weights of this best model as `q1_model.pt`.

4. 5 points for visualizing the filters. 5 points for describing what you observe from it.

5. 5 points for the bar plots corresponding to filters from the first and the last layer. 5 points for explaining what you can say from those plots about the difference between the first and the last layers in terms of what they learn.

References

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems*. 2012.

[ZF14] Matthew D Zeiler and Rob Fergus. "Visualizing and understanding convolutional networks". In: *European Conference on Computer Vision*. 2014.