

**Project 2: Steven's Convoluted Quest to See the World and Stay Cool**  
**An exploration of deep learning techniques for classification and feature learning**  
**Due: Thurs. 03/21 (11:59pm)**

## Introduction

After hearing about the power of classification from EECS 445 students, Steven was able to find which airline is truly the best. He is now embarking on his post-graduation trip, flying all over the world in hopes of widening his perspective. More importantly, Steven aims to finally reach 100 Instagram followers by posting pictures of monuments with cute hashtags. There's just one problem: Steven won't be able to come up with clever posts unless he knows the names of the monuments!

Our job is to help Steven on his spiritual journey by determining the name of the monument in each of his photos. Based on his itinerary, we have gathered training data on 10 monuments that we want to classify: Colosseum, Petronas Towers, Rialto Bridge, Museu Nacional d'Art de Catalunya (MNAC), Pantheon, Hofburg Imperial Palace, Berlin Cathedral, Hagia Sophia, Gaudi's Casa Batllo, and St. Stephen Cathedral. Some of these images are shown below:



Figure 1: Sample images from the the dataset.

In this project, we will explore both supervised and unsupervised deep learning techniques for image data. In the supervised setting, we will use a deep convolutional neural network (CNN) to classify Steven's photos by monument. In the unsupervised setting, we will work exclusively with the image data to learn a feature representation (i.e., a continuous-valued feature vector) for each input image. We will then explore *unsupervised pretraining*, a type of *transfer learning* that will use our unsupervised feature representation to augment our classification abilities in the supervised setting. If you are unsure about what some of these terms mean — don't worry! We will be exploring these ideas throughout this project. After this, you will be well-equipped to take on Steven's challenge: implementing and training a deep neural network of your own design to identify monuments in images!

# Getting Started

## Skeleton Code

Please download the Project 2 skeleton code `project2.zip` from Canvas. This zip file contains the full project dataset, and may take several minutes to download. After unzipping, please ensure the all of the following files are present:

- **Executables**

- `evaluate_autoencoder.py`
- `predict_challenge.py`
- `train_autoencoder.py`
- `train_autoencoder_classifier.py`
- `train_challenge.py`
- `train_cnn.py`
- `visualize_autoencoder.py`
- `visualize_cnn.py`
- `visualize_data.py`

- **Models**

- `model/autoencoder.py`
- `model/challenge.py`
- `model/cnn.py`

- **Checkpoints**

- `checkpoints/autoencoder/`
- `checkpoints/autoencoder_classifier/`
- `checkpoints/challenge/`
- `checkpoints/cnn/`

- **Data**

- `data/landmarks.csv`
- `data/images/`
- `dataset.py`


- **Utilities**

- `config.json`
- `train_common.py`
- `utils.py`

## Dataset

This dataset contains 12,500 PNG image files of 10 different monuments. These images are named as `000000.png` through `012499.png`, and stored under `data/images/`. Each image contains 3 color channels (red, green, blue) and is of size  $128 \times 128 \times 3$ . These images have already been divided into 3 class-balanced data partitions: a training set, a validation set, and a test set. The metadata containing the label and data partition for each image is documented in `data/landmarks.csv`. Note that the test labels have been removed from this file. As in project 1, you will include your challenge model's predictions on this data partition as a part of your submission.

## Python Packages

This project will introduce you to  PyTorch, a deep learning framework written and open-sourced by Facebook's AI group. To install PyTorch, follow the instructions on <https://pytorch.org/get-started/locally/>. We recommend selecting Pytorch Build - Stable and CUDA - None. At the time of writing this document, the latest stable version is 1.0. We have written this project to run in a reasonable time on standard laptop machines; however, if you have access to a compatible GPU and wish to use it, please select the appropriate CUDA version. To ensure fairness, challenge submissions that utilize GPU hardware will be graded separately from non-GPU submissions.

Besides PyTorch, there is one additional package new to this assignment: [Pillow](#). This package serves as the backend for a number of useful image processing techniques in `scipy`, such as resizing. Install it by running the command `pip install Pillow`.

## 1 Data Preprocessing [10 pts]

Real datasets often contain numerous imperfections (e.g., wrong labels, noise, or irrelevant examples) and may not be immediately useful for training our model. To develop a dataset more conducive for training, we will explore a few image preprocessing techniques. These will serve two purposes: speeding up computation time and highlighting the underlying structure of the image dataset. These preprocessing steps will be implemented in the file `dataset.py`.

- (a) The first step of preprocessing is to downsample (reducing the dimensions) our images. We make the assumption that the high-frequency information in our images is not especially useful for distinguishing among landmarks, and can be discarded. Regardless of whether this assumption is correct, the downsampling will also greatly improve the speed at which our model will train and perform predictions. **Implement** the `resize()` function by using the imported `imresize` function to transform the images from their original dimensions ( $128 \times 128 \times 3$ ) to size `image_dim × image_dim × 3`. Set the argument `interp='bicubic'` to use bicubic interpolation. Here `image_dim` is the dimension we define in the `config.json` file and retrieve via the `config()` helper function. As we will use this helper function to retrieve program arguments and hyperparameters throughout this project, it is worth taking a moment to understand how this is accomplished.
- (b) The next step is to standardize our images, so their features have a similar range of values. **Implement** the `fit()` and `transform()` functions in the `ImageStandardizer` class. The `fit()`

function is first called with the training data set as input, and it calculates and stores the per-channel mean and standard deviation from the training data. Specifically, we assume that the underlying distributions of each image channel (red, green, and blue) are independent distributions with mean 0 and variance 1. To embed these assumptions, zero-center each image channel by subtracting the per-channel mean value, then scale each image channel by dividing by the per-channel standard deviation. These mean and standard deviation values are applied to the standardization of all data partitions with the `transform()` function. **Answer the following question regarding this normalization process.**

- i. Run the script `dataset.py`, and report the mean and standard deviation of each color channel (RGB), learned from the entire training partition.
  - ii. Why do we extract the per-channel image mean and standard deviation from the training set as opposed to the other data partitions?
- (c) **Run** the script `visualize_data.py` to see the effects of this preprocessing on a couple of example images. **Save and include** the resulting plots in your write-up. **What** are the visible effects of our preprocessing on the image data?

## 2 Convolutional Neural Networks [30 pts]

With our data now in a suitable form for training, we will explore the implementation and effectiveness of a convolution neural network (CNN) on the dataset. For this question, we will focus to the **first 5 classes**: Colosseum, Petronas Towers, Rialto Bridge, MNAC, and Pantheon. Recall the basic structure of a convolutional network: a sequential combination of convolutional, activation, pooling and fully connected layers.

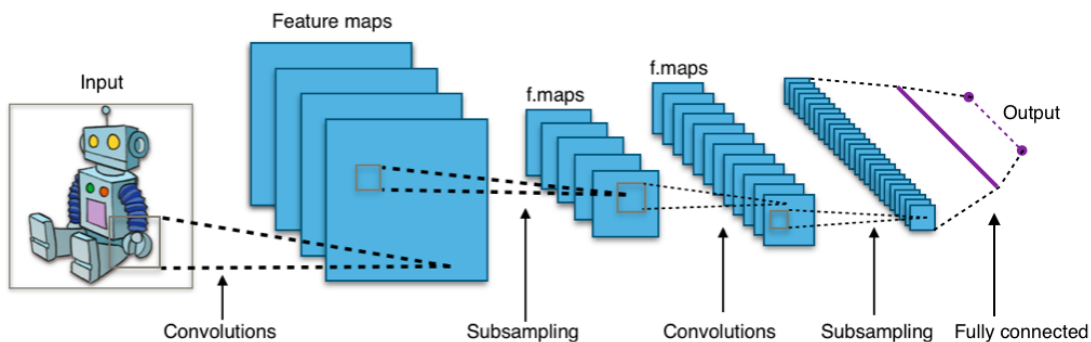


Figure 2: An example convolutional neural network. By Aphex34, licensed under [CC BY-SA 4.0](#).

CNNs used for image classification can often be viewed in two parts, where the first part learns a representation of the image data — or “encoding” — and the second learns a mapping from this image encoding to the output space. Our CNN will follow this schema, and the output space will be the five class labels we are considering. The image encoder will be composed of convolutional layers, which maintain the spatial structure of the image data by sliding filters with learned weights across the image. Once we have our image encoding, we will use fully connected layers to learn a non-linear combination of this representation, and

ideally map this representation into a linearly-separable space that we can trivially classify. Between layers, we will add non-linear activation functions that allow the network to learn a non-linear mapping.

We have provided the training framework necessary to complete this problem in the file `train_cnn.py`. All of the following problems that involve implementation will require you to use the PyTorch API. We've provided some tips in Appendix A, and you are encouraged to explore the [documentation](#) and [online tutorials](#) to learn more.

- (a) **Study** the CNN architecture specified in Appendix B (on page 13). **How many learnable float-valued parameters does the model have?** You should be able to manually calculate this number, and later verify against the program output after implementing the model.
- (b) **Implement** the architecture by completing the `CNN` class in `model/cnn.py`. In PyTorch, we define a neural network by subclassing `torch.nn.Module`, which handles all the gradient computations automatically. Your job is to fill in the appropriate lines in three functions: `__init__()`, `init_weights()` and `forward(x)`.
  - Use `__init__()` to define the architecture, i.e. what layers our network contains. At the end of `__init__()` we call `init_weights()` to initialize all model parameters (weights and biases) in all layers to desired distributions.
  - The `forward(x)` function defines the forward propagation for a batch of input examples, by successively passing output of the previous layer as the input into the next layer after applying activation functions, and returning the final output as a `torch.Tensor` object.
  - The `torch.Tensor` class implements a `backward()` function. As its name suggests, it performs back propagation and computes the partial derivatives with respect to each model parameter using chain rule. While you do not have to implement `backward()`, you should review the lecture slides on backpropagation, so that you understand why it is important when training the network.
- (c) After making a forward pass we can use the output from the CNN to make a prediction. **Implement** the `predictions()` function in `train_common.py` to predict the numeric class label given the network output logits. These logits are just the raw output predictions of the network (without normalization).
- (d) **Review** the trainer script `train_cnn.py`. Based on the model specification in Appendix B, **fill in** the definitions for `criterion` and `optimizer`. In the main function, we have a training loop that loops through the dataset for a fixed number of times as defined in the config file. After each complete pass of the training set (called an “epoch”), we evaluate the model on the validation set, and save this solution as a checkpoint. **Review** the definitions of the following functions:
  - `_train_epoch`: within one epoch, we pass batches of training examples through the network, use back propagation to compute gradients, and update model weights using the gradients.
  - `_evaluate_epoch`: we pass the entire validation set (in batches) through the network and get the model's predictions, and compare these with the true labels to get an evaluation metric.
  - `save_checkpoint`: checkpointing — the periodic saving of model parameters — is an important technique for training large models in machine learning; if a hardware failure occurs due to a power outage or our code fails for whatever reason, we don't want to lose all of our progress!

- (e) **Execute** `train_cnn.py` to train the model.

Note that if you run the script more than once, you will be prompted to enter the epoch number at which the model parameters from the saved checkpoint will be restored and training will *continue* from that epoch. If you enter 0, then the model will be trained from scratch and the old checkpoints will be deleted.

This script will also create two graphs that are updated every epoch to monitor the model's performance in terms of loss and accuracy on the train and validation set. **Include the final graphs in your write-up and answer the following questions.**

- i. You should observe that the training plot is fairly noisy and validation loss does not monotonically decrease. Describe at least two sources of noise that contribute to this behavior.
  - ii. If we were to continue training the model, what do you expect will happen to (1) training loss and (2) validation loss? Consider both the effects of additional training on these individual statistics as well as their relation to each other.
  - iii. Here, we stopped training after a fixed number of iterations. Based on your training plot, at which epoch should you stop training the model? Write down this value, and your reasoning for why you picked this value. Use this epoch number for the next problem.
- (f) In a convolutional layer, every filter has different weights and generates a distinct activation map given the same input. Activation maps represent the output of a convolutional layer. They are useful because they show us how well a particular filter matched individual patches of the input image. For example, the first convolutional layer has 16 filters, thus we have 16 different activation maps. These activation maps are single-channel images, and we can plot them using a continuum of colors, where darker color indicates small values and lighter color indicates large values. Filters in one layer are sorted by the mean of each filter's weights to improve consistency of results. By visualizing the activation maps, we are able to see what each filter is learning.

**Execute** `visualize_cnn.py` to generate visualizations for the activation maps of the first convolutional layer, when each of the following three images is passed through the network. These sample images are representative of their respective classes. **Include the plots in your write-up.**

**Answer the following questions:**

- i. **What** do you notice about the first-layer activations for input images of different classes? Compare and contrast the output from these three images based on their similarities and differences (in terms of color, shape, texture, etc.).
- ii. **What** low-level features could each filter be looking for?



00013.png  
 $y = 3$   
MNAC



00014.png  
 $y = 4$   
Pantheon



00020.png  
 $y = 0$   
Colloiseum

### 3 Classification by Autoencoder [40 pts]

In many cases, we may have a large amount of data, but there are no corresponding labels. Such unlabeled data may still be useful if we can find a way to learn a good representation of the underlying structure. The general task of this form, called **unsupervised learning**, involves learning NOT how to predict the output label given the input feature-vector, but instead the internal structure of the input relative to other data points. There are many architectures and training methods for doing unsupervised learning with neural nets; here, we'll focus on **autoencoders**.

We consider a good representation of our data to be a compact representation in a lower-dimensional space, since it would require our model to learn the simpler underlying structure of the data instead of memorizing specific features of the data. Specifically, if  $\mathcal{X}$  is a space of possible landmark images, we want to learn a  $d$ -dimensional representation of the observed elements of  $\mathcal{X}$ , where  $d$  is much smaller than the dimensions of  $\mathcal{X}$ . To learn such a representation, we can employ a function encoder :  $\mathcal{X} \rightarrow \mathbb{R}^d$ , which maps input images to the  $d$ -dimensional representation. However, without a semantic representation of the important features (e.g. labels), we cannot directly train a network to learn the function encoder.

Instead of requiring labels, let's utilize what we do have: data. Instead of learning encoder :  $\mathcal{X} \rightarrow \mathbb{R}^d$  directly, we can learn an identity function  $\text{ident} : \mathcal{X} \rightarrow \mathcal{X}$ . Now, we have both the inputs and the outputs, so we can learn this function. We seek to approximate the identity function by a functional form that compresses and then decompresses the input data. We can break this function down into two parts:  $\text{ident} = \text{decoder} \circ \text{encoder}$ , where encoder :  $\mathcal{X} \rightarrow \mathbb{R}^d$  and decoder :  $\mathbb{R}^d \rightarrow \mathcal{X}$ . Our training pairs for learning this function will be of the form  $(\bar{x}, \bar{x})$ , where  $\bar{x}$  is a landmark image. Intuitively, the learned compressor (encoder) will throw away all irrelevant aspects of the data and keep only what is essential for the decompressor (decoder) to reconstruct the data. This will allow us to find a good representation of the data.

A neural network that implements this idea is called an **autoencoder**. Note: in general, the performance of a neural network increases significantly with the amount of data. Thus, we will use **all 10 classes** from the **entire available dataset** for this part of the project: 7500 training images and 2500 test images (even though we don't have labels for them). The remaining 2500 validation images will be used to evaluate the performance of the autoencoder.

- (a) **Study** the architecture presented in Appendix C (page 15).

Notice the “deconvolutional” layer. This layer is simply a convolutional layer whose output layer has larger side-length than the input. While the convolutional layer shrinks images, the deconvolutional layer grows them.

**Answer the following in your write-up:**

- i. Which layers have weights corresponding to encoder, and which to decoder?

Observe a rough symmetry between encoder and decoder: such symmetries are common in the design of autoencoders, since the encoder and decoder are intended to be rough inverses. In designing an autoencoder, we are faced with many architecture choices.

- ii. Speculate on these architecture choices by **completing** the table below (use your intuition). Compare using the alternative choice against using the current choice, and fill in either  $\uparrow$  or  $\downarrow$  in



the right three columns. For each pair of current and alternative choices, write a **short justification** that **identifies the main differences**. For the last two rows, fill in the current choice based on our architecture and your own choice of an alternative.

	Current choice	Alternative	How might the alternative choice affect...		
			Training Speed	Structural Error	Estimation Error
Initialization	random normal	zero	↓	↑	
Activation	ELU	Sigmoid			
Depth					
Regularization					

- (b) **Implement** the architecture by filling out the `Autoencoder` class in `model/autoencoder.py`. Some of the layers are already implemented for you. You may change the given lines if necessary, but a simple and correct solution can be built solely by adding to the given lines. Your job is to fill in the appropriate lines in four functions: `__init__()`, `init_weights()`, `encoder(x)` and `decoder(x)`.
- The `__init__()` and `init_weights()` functions have the same functionality as before.
  - The `encoder(x)` function defines the operations for the encoder part of the autoencoder, which maps the input examples to  $d$ -dimensional representations.
  - The `decoder(x)` function defines the operations for the decoder part of the autoencoder, which maps the  $d$ -dimensional representations from the `encoder` back to the original images.
  - The `forward(x)` function defines the forward propagation and is implemented for you. It simply composes the `encoder(x)` and `decoder(x)` functions.
- (c) **Review** the script `train_autoencoder.py`. Based on the model specification in Appendix C, **fill in** the definitions for `criterion` and `optimizer`. **Complete** the `train_epoch` function based on instructions in the skeleton code. Review the `evaluate_epoch` function.
- (d) **Run** the script `train_autoencoder.py` to train the network on the training data and evaluate on the validation set. On a standard laptop, this should take about 3 minutes. **Include the final training plot of validation MSE. Report the stopping epoch number and use this value for the next few problems.**
- (e) **Execute** the script `evaluate_autoencoder.py` and **communicate** the network performance on the validation set as follows.
- Report a baseline reconstruction error of your choice. For instance, what is the reconstruction mean squared error of the compression-decompression scheme that always returns the same reconstruction: a uniformly gray square?
  - Report the reconstruction error overall, and for each image class.
  - Display the `ae_per_class_perf.png` image saved to your project directory, which shows the best/worst/typical reconstructions for each class. What kind of image is the autoencoder best at reconstructing?



- (f) **Run** the script `visualize_autoencoder.py`. What aspects of landmarks does the autoencoder seem to have learned? **Qualitatively evaluate** the autoencoder's performance as a compression-reconstruction method, support your evaluation with the `ae_recon_comparison.png` image saved to your project directory. You should **contrast** the autoencoder's output with that of a naive downsample-then-upsample method with matching compression ratio.
- (g) Instead of randomly initializing model weights, we can initialize them with weights obtained from training on a different (but related) task. This allows our model to utilize knowledge learned on another task for our current task, so our model does not have to learn everything from scratch. Usually, this also leads to faster convergence. This is the idea behind **transfer learning**.

Here, you will use transfer learning to train a landmark classifier similar to the CNN in Q2, but now making use of the encoder of the autoencoder. You will freeze the weights of the autoencoder, connect the output of the encoder to more fully-connected layers, and then train these last layers on the classification task. **Fill in** the missing lines in the `AutoencoderClassifier` model in `autoencoder.py` (your implementation should be the same as the encoder of your `Autoencoder`). **Review** and **run** the script `train_autoencoder_classifier.py` to train our classifier to classify the first five classes of landmarks. **Compare** this classifier to your previous CNN classifier by reporting the accuracy on the validation set for each class (you will have to implement this yourself). Format your results as follows:

	CNN accuracy	Autoencoder classifier accuracy
$y = 0$ Colosseum		
$y = 1$ Petronas Towers		
$y = 2$ Rialto Bridge		
$y = 3$ MNAC		
$y = 4$ Pantheon		

**How** does this classifier compare to the CNN in Q2? Comment on performance and training time.

## 4 Challenge [20 pts]

Armed with your knowledge of neural networks and PyTorch, you are now ready to help Steven: **designing**, **implementing**, and **training** a deep neural network of your own design to classify landmarks! We will again restrict our final classifier to the first 5 classes; however, you will now be allowed to use the training and validation sets for all 10 classes to train your model. These additional data may be useful if you wish to further explore transfer learning in your challenge solution. In addition, you will have *full control* over the network architecture, including weight initialization, filter sizes, activation function, network depth, regularization techniques, preprocessing methods, and any other changes you see fit. We ask you to adhere to the following guidelines when developing your challenge submission.

- You *must* use only the data provided for this challenge; *Do not attempt to acquire any additional training data, or the ground truth test labels. The use of pretrained models is also not allowed.*
- **Implement** your model within `model/challenge.py`.
- **Fill in** the definitions for the loss function, optimizer, model, and the `train_epoch` function.
- You may alter the training framework in `train_challenge.py` as you please.
- If you wish to modify any portion of the data batching or preprocessing, please make a copy of `dataset.py` and call it `dataset_challenge.py`. You may then modify this file however you choose.
- You may add additional variables to the `config.json` file and modify any of the existing challenge variables as you see fit.

Once your model has been trained, ensure that its checkpoint exists in the directory `checkpoints/challenge/` and **run** the script `predict_challenge.py` as follows.

```
python predict_challenge.py --username=<your_username>
```

This script will load your model from your saved checkpoint and produce the file `<your_username>.csv` that contains your model's predictions on the test set.

In your write-up, describe the experiments you conducted and how they influenced your final model. When grading your challenge write-up, we will look for at least some discussion of your decisions regarding the following.

- Regularization (weight decay, dropout, etc.)
- Feature selection
- Model architecture
- Hyperparameters
- Model evaluation (i.e., the criteria you used to determine which model is best)

- Whether your model was trained using GPU hardware

We will evaluate your submission based on two components with the points split evenly between the two:

1. Effort (10 pts): We will evaluate how much effort you have applied to this problem based on the experiments described in your write-up and your code submission.
2. Accuracy (10 pts): We will evaluate the top-1 accuracy of your classifier's predictions on the ground truth test labels. In other words, given  $n$  test images, ground truth test labels  $\{y^{(i)}\}_{i=1}^n$ , and your predictions  $\{\hat{y}^{(i)}\}_{i=1}^n$ , your performance will be evaluated as follows.

$$\text{accuracy} = \frac{1}{n} \sum_{i=1}^n [[y^{(i)} == \hat{y}^{(i)}]]$$

Note that this means that the output test predictions follow a specified order. For that reason, please do not shuffle the test set data points during prediction.

**REMEMBER to submit your project report by 11:59pm ET on March 21st, 2019 to Gradescope.**

**Include** your code as an appendix (copy and pasted) in your report. Please try to format lines of code so they are visible within the pages.

**Upload** your file `username.csv` containing the label predictions for the test images at this link <https://tinyurl.com/eecs445p2>, providing your [umich.edu](https://umich.edu) email.

## Appendix A Implementation notes

### A.1 `torch.nn.Conv2d`

This class implements the 2D convolutional layer we have learned in lecture. Create the layer as follows: `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)`. Use default settings for all other arguments. For example `dilation=1` means no dilation is used, and `bias=True` adds a learnable bias term to the neuron.

### A.2 the SAME padding

With Pytorch's default setting `padding=0`, if we apply a  $3 \times 3$  convolutional filter to a  $4 \times 4$  input image, the output would be  $2 \times 2$ . As we keep applying convolutional layers in this way, the spatial dimensions will keep decreasing. However, in the early layers of a CNN, we want to preserve as much information about the original input volume so that we can extract those low level features. To do this, we can pad the borders of the input image with zeros in a way such that the output dimension is the SAME as the input (assuming unit strides). If the filter size is odd  $k = 2m + 1$ , then this amount of zero padding on each side is  $p = \lfloor k/2 \rfloor = m$ . For example, in Figure 3, since the filter size is  $k = 3$ , the appropriate padding size is  $p = \lfloor 3/2 \rfloor = 1$ .

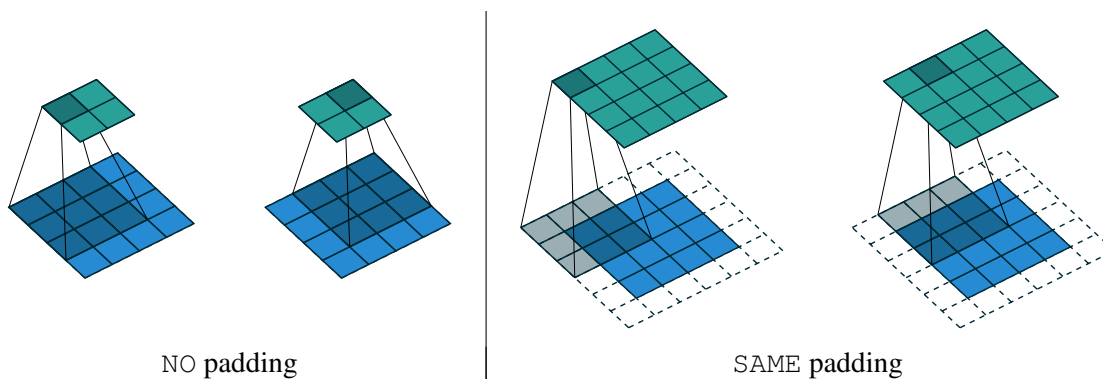


Figure 3: Comparison of padding schemes.

Adapted from: Vincent Dumoulin, Francesco Visin - A guide to convolution arithmetic for deep learning.

Source code: [https://github.com/vdumoulin/conv\\_arithmetic](https://github.com/vdumoulin/conv_arithmetic)

### A.3 `torch.nn.CrossEntropyLoss`

Let  $\bar{z}$  be the network output logits and  $\hat{y}_c$  the probability that the network assigns to the input example as belonging to class  $c$ . Let  $\bar{y} \in \mathbb{R}^D$  be a one-hot vector with a one in the position of the true label and zero otherwise (i.e., if the true label is  $t$ , then  $y_c = 1$  if  $t = c$  and  $y_c = 0$  if  $t \neq c$ ).  $D$  is the number of classes.

For a multiclass classification problem, we typically apply a softmax activation at the output layer to generate a probability distribution vector. Each entry of this vector can be computed as:

$$\hat{y}_c = \frac{\exp(z_c)}{\sum_j \exp(z_j)}$$

We then compare this probability distribution  $\hat{y}$  with the ground truth distribution  $\bar{y}$  (a one-hot vector), computing the cross entropy loss,  $\mathcal{L}$ :

$$\mathcal{L}(\bar{y}, \hat{y}) = - \sum_c y_c \log \hat{y}_c$$

These two steps can be done at once in PyTorch using the `CrossEntropyLoss` class, which combines the softmax activation with negative log likelihood loss. We don't need to add a separate soft max activation at the output layer. This improves computational efficiency and numerical stability, and you will explore more about the mathematical reason in Homework 3.

## Appendix B CNN Architecture

### Training Parameters

- Criterion: `torch.nn.CrossEntropyLoss`
- Optimizer: `torch.optim.Adam`
- Learning rate:  $10^{-4}$
- Number of epochs: 40
- Batch Size: 128

### Architecture

Layer 0: Input image

- Output:  $3 \times 32 \times 32$

Layer 1: Convolutional Layer 1

- Number of filters: 16
- Filter size:  $5 \times 5$
- Stride size:  $2 \times 2$
- Padding: SAME
- Activation: ReLU
- Weight initialization: normally distributed with  $\mu = 0.0$ ,  $\sigma^2 = \frac{1}{5 \times 5 \times 3}$
- Bias initialization: constant 0.0
- Output:  $16 \times 16 \times 16$

Layer 2: Convolutional Layer 2

- Number of filters: 64

- Filter size:  $5 \times 5$
- Stride size:  $2 \times 2$
- Padding: SAME
- Activation: ReLU
- Weight initialization: normally distributed with  $\mu = 0.0$ ,  $\sigma^2 = \frac{1}{5 \times 5 \times 16}$
- Bias initialization: constant 0.0
- Output:  $64 \times 8 \times 8$

Layer 3: Convolutional Layer 3

- Number of filters: 32
- Filter size:  $5 \times 5$
- Stride size:  $2 \times 2$
- Padding: SAME
- Activation: ReLU
- Weight initialization: normally distributed with  $\mu = 0.0$ ,  $\sigma^2 = \frac{1}{5 \times 5 \times 64}$
- Bias initialization: constant 0.0
- Output:  $32 \times 4 \times 4$

Layer 4: Fully connected layer 1

- Input: 512
- Activation: ReLU
- Weight initialization: normally distributed with  $\mu = 0.0$ ,  $\sigma^2 = \frac{1}{512}$
- Bias initialization: constant 0.0
- Output: 64

Layer 5: Fully connected layer 2

- Input: 64
- Activation: ReLU
- Weight initialization: normally distributed with  $\mu = 0.0$ ,  $\sigma^2 = \frac{1}{64}$
- Bias initialization: constant 0.0
- Output: 32

Layer 6: Fully connected layer 3 (Output layer)

- Input: 32
- Activation: None
- Weight initialization: normally distributed with  $\mu = 0.0$ ,  $\sigma^2 = \frac{1}{32}$
- Bias initialization: constant 0.0
- Output: 5 (number of classes)

## Appendix C Autoencoder Architecture

### Training Parameters

- Criterion: `torch.nn.MSELoss`
- Optimizer: `torch.optim.Adam`
- Learning rate:  $10^{-4}$
- Number of epochs: 20
- Batch Size: 128

### Architecture

Layer 0: Input image

- Output:  $3 \times 32 \times 32$

Layer 1: Average Pooling

- Filter size:  $2 \times 2$
- Stride size:  $2 \times 2$
- Output:  $3 \times 16 \times 16$

Layer 2: Fully connected layer

- Input: 768
- Activation: ELU
- Weight initialization: normally distributed with  $\mu = 0.0$ ,  $\sigma^2 = \frac{0.1^2}{768}$
- Bias initialization: constant 0.01
- Output: 128

Layer 3: Fully connected layer

- Input: 128
- Activation: ELU
- Weight initialization: normally distributed with  $\mu = 0.0$ ,  $\sigma^2 = \frac{0.1^2}{128}$
- Bias initialization: constant 0.01
- Output: 64

Layer 4: Fully connected layer

- Input: 64
- Activation: ELU



- Weight initialization: normally distributed with  $\mu = 0.0$ ,  $\sigma^2 = \frac{0.1^2}{64}$
- Bias initialization: constant 0.01
- Output: 20736

Layer 5: Deconvolutional Layer (implemented for you)

- Input:  $64 \times 18 \times 18$
- Activation: Identity
- Weight initialization: normally distributed with  $\mu = 0.0$ ,  $\sigma^2 = 0.01^2$
- Bias initialization: constant 0.00
- Filter size:  $5 \times 5$
- Padding: SAME
- Output:  $3 \times 36 \times 36$

Layer 6: Centered Crop (implemented for you)

- Output:  $3 \times 32 \times 32$

Layer 7: Image-by-Image Color Normalization (no parameters; implemented for you)

- Target pixel-value mean: 0.0
- Target pixel-value standard deviation: 1.0
- Output:  $3 \times 32 \times 32$

## Appendix D Expected Timeline

This section will describe the expected time of completion along with an estimated workload (in number of days) for each section of the project. This should give you a sense of whether you are on schedule to complete the project.

- 1. Data Preprocessing:** March 11th, 2019 (1 day).
- 2. Convolutional Neural Networks:** March 15th, 2019 (3 days).
- 3. Classification by Autoencoder:** March 18th, 2019 (3 days).
- 4. Challenge:** March 21th, 2019 (3 days).