Project: Classifying AHE with CNN

- Sajjad Zangiabadi

Dataset Description:

- The images included in the Architectural Heritage Elements dataset are licensed under Creative Commons and are mainly taken from Flickr and Wikimedia Commons. Ten categories—Apse, Bell tower, Column, Inner and Outer Dome, Flying buttress, Gargoyle, Stained glass, and Vault—have been systematically assigned to these photos. This system of classification captures the variety of architectural features that are frequently found in historic buildings.

Dataset summary:

- Dataset Name: Architectural Heritage Elements Image64 Dataset
- Dataset Type: Image dataset
- Number of Images: 10235
- Number of Categories: 10
- Categories: Altar, Apse, Bell tower, Column, Dome (inner), Dome (outer), Flying buttress, Gargoyle, Stained glass, Vault
- Source: Flickr and Wikimedia Commons
- License: Creative Commons

Data Preprocessing:

- Function “convert_to_dataframe”:
  - This function receives as inputs the path to the dataset's root folder and a mapping dictionary which links class names to labels. Using the folder names as a guide, iteratively traversing the directory structure, it extracts picture paths and labels that go with them. Eventually, a Pandas DataFrame with columns called "path" and "label" is created and returned.
- Class “CustomDataset”:
  - This class, which takes its inheritance from the Pytorch Dataset class, is made to deal with loading and preparing images for training. Its input consists of the dataset root directory, a Pandas DataFrame with picture paths and labels, and an optional transformation object. The number of photos in the dataset is returned by the __len__() method. At a given index, the __getitem__() function returns an image together with its associated label. The label is first taken out of the DataFrame and transformed into a PyTorch tensor. Next, it uses the root directory and path data from the DataFrame to create the entire picture path. Pillow (PIL Fork) is used to load the image and convert it to RGB representation. Lastly, before returning a pair, a transformation object, if one is supplied, is applied to the image.

Training Function:

- The function uses the given dataloader to iterate over batches of data.
  It transfers the labels and the input data to the proper computing equipment (such as a GPU).
  To enable gradient computing, the model is in training mode.
  The model produces predictions for each batch and uses the given criterion to compute the loss.
  To track training error, the loss is added up.
  In order to monitor training accuracy, the function calculates the number of accurate predictions.
  To update the model parameters, optimization via gradient descent takes place out.

Evaluating Funciton:

- The function is used to assess the performance of the CNN model. Like the training loop, iterations are performed over batches of images and labels. The main difference is the torch.no_grad() context, which suspends gradient computation during evaluation for efficiency. The evaluation set's average loss and accuracy are calculated by the function, which provides insight into how well the model generalizes to new data.

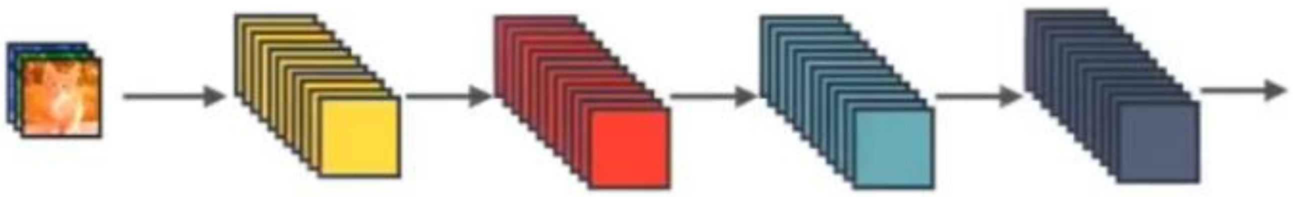
Run Over Epochs:

- The training process is managed over several epochs by the run_over_epoches function. It keeps track of training and evaluation losses and accuracy over epochs by maintaining a history dictionary. It also maintains track of the top-performing model according to its validation accuracy within each epoch.

Image Preprocessing with Transformations

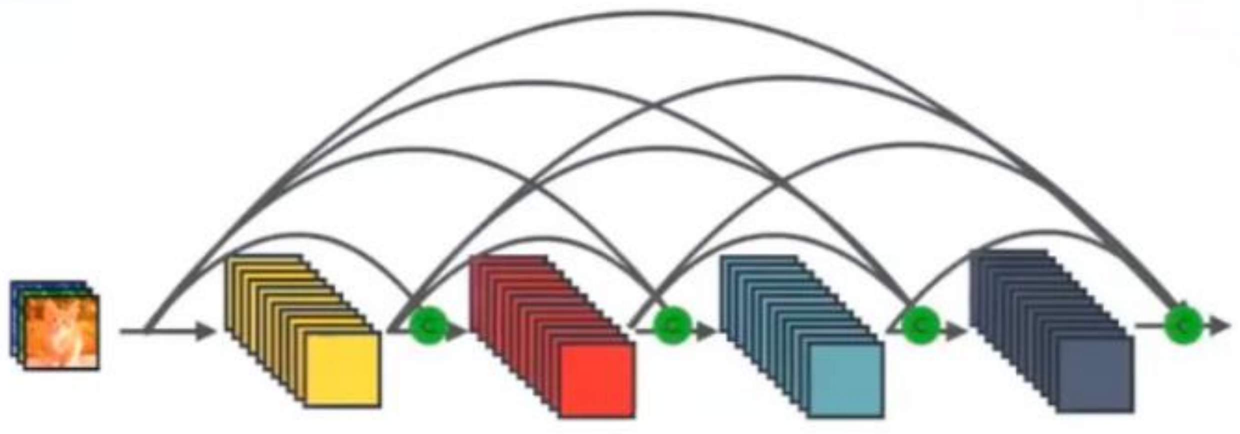
- **Purpose:** Transformations are employed to:
  - **Augment Data:** Techniques like random flipping, cropping, and rotation artificially increase the dataset size and introduce variations in image appearance. This improves the model's capacity to generalize to previously unknown data by teaching it properties that are invariant to small changes in views or object placements.
  - **Normalize Data:** usually, normalization involves scaling pixel values to a predetermined range (usually, between 0 and 1) and perhaps centering the distribution around a predetermined mean value. This keeps certain channels from controlling the learning process and guarantees that the scales of various image channels are comparable.
- **Specific Transformations (for training set):**
  - **ToTensor()**: transforms PIL pictures into PyTorch tensors with pixel values normalized between 0 and 1
  - **RandomHorizontalFlip(p=0.2)**: Randomly flips images horizontally with a probability of 0.2, introducing viewpoint variations.
  - **RandomVerticalFlip(p=0.2)**: Randomly flips images vertically with a probability of 0.2, introducing viewpoint variations.
  - **RandomResizedCrop((224, 224), antialias=True)**: Randomly crops a resized patch of size (224, 224) from the original image, addressing variations in image scale and composition.
  - **RandomRotation(15)**: Randomly rotates the image by up to 15 degrees, simulating slight viewpoint changes.
  - **Normalize([mean], [std])**: Using pre-calculated statistics from the training dataset, each color channel is normalized by taking the mean and dividing by the standard deviation of the channel. This guarantees that during training, every channel contributes equally.
- **Separate Transformations:**
  - The code defines three distinct transformations train_transform, dev_transform, and test_transform with slightly different normalization parameters mean and std. This is a typical procedure. Different transformations are used for training validation and test dataset. The number and diversity of training data are increased by augmentations (flips, crops, and rotations), and for all datasets are standardized using the data's statistics of themself.

Densenet



Standard ConvNet Concept

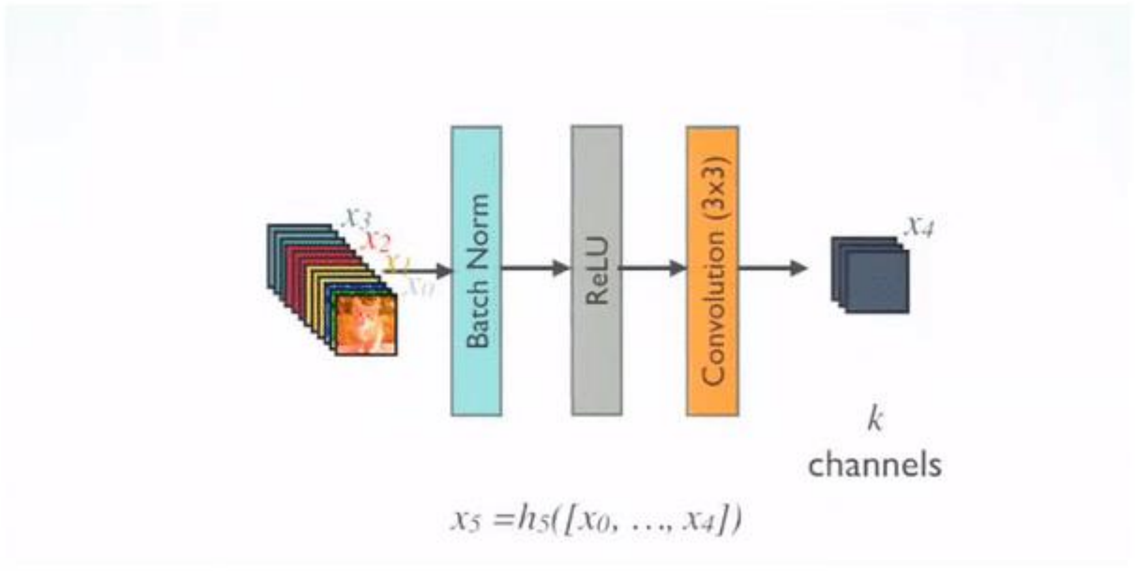
In **Standard ConvNet**, input image goes through multiple convolution and obtain high-level features.



One Dense Block in DenseNet

In **DenseNet**, each layer obtains additional inputs from all preceding layers and passes on its own feature-maps to all subsequent layers. **Concatenation** is used. **Each layer is receiving a "collective knowledge" from all preceding layers.**

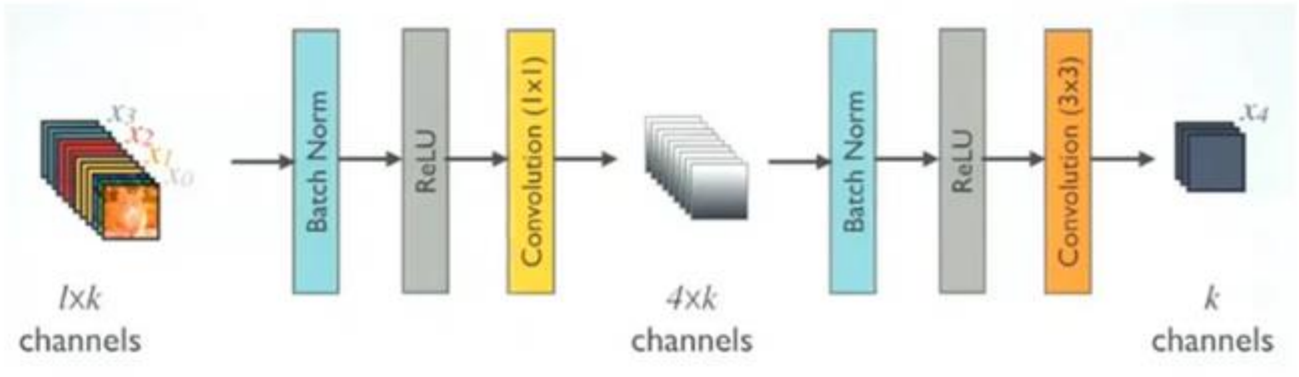
2.1. Basic DenseNet Composition Layer



Composition Layer

For each composition layer, **Pre-Activation Batch Norm (BN) and ReLU**, then **3×3 Conv** are done with output feature maps of *k* channels, say for example,

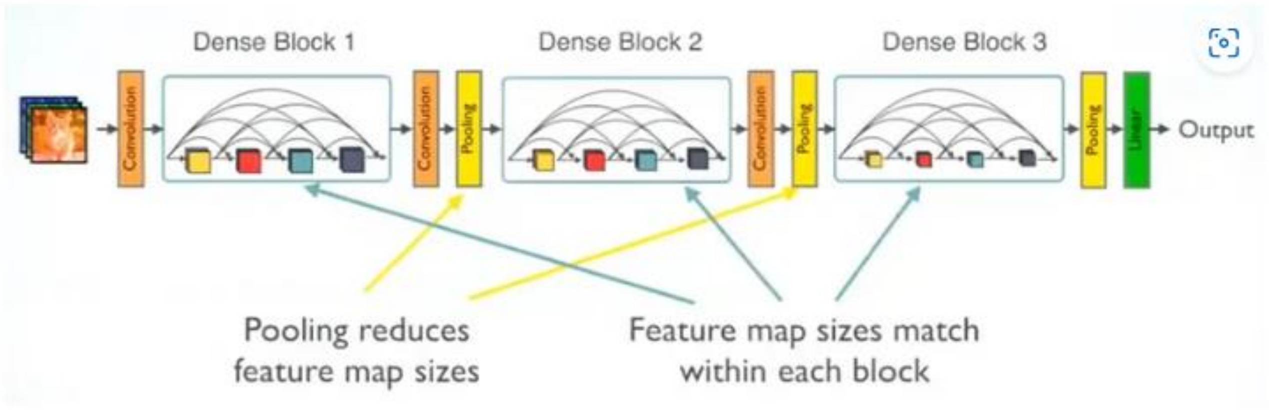
2.2. DenseNet-B (Bottleneck Layers)



DenseNet-B

To reduce the model complexity and size, **BN-ReLU-1×1 Conv** is done before **BN-ReLU-3×3 Conv.**

2.4. Multiple Dense Blocks with Transition Layers


Dense Block 1
Dense Block 2
Dense Block 3
Convolution
Pooling
Linear
Output
Pooling reduces feature map sizes
Feature map sizes match within each block

Multiple Dense Blocks

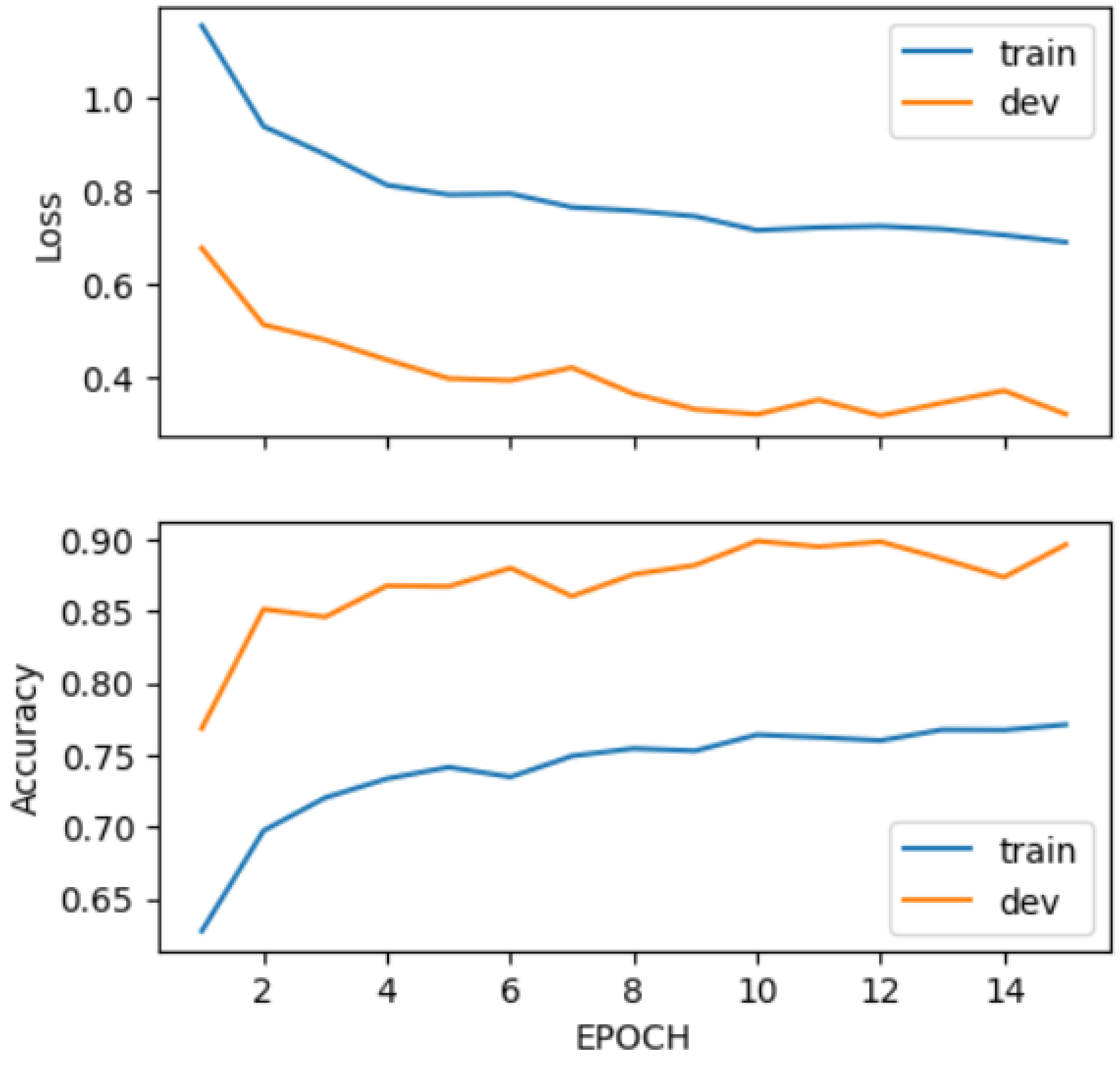
1×1 Conv followed by 2×2 average pooling are used as the transition layers between two contiguous dense blocks.

Feature map sizes are the same within the dense block so that they can be concatenated together easily.

At the end of the last dense block, a global average pooling is performed and then a softmax classifier is attached.

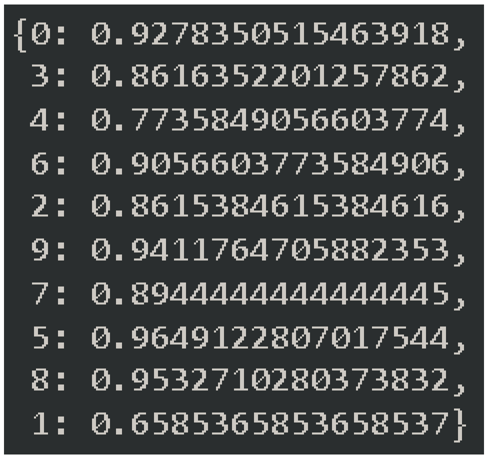
Training history

- A single plot with two subplots is used by the plot_history function to display the training process. The training epochs are represented by the x-axis.
- Loss Subplot: The training loss (blue line) and validation loss (orange line) over epochs are displayed in the top subplot. As the model learns, the training loss ideally gradually drops. The validation loss is used as a generalization indicator and this example shows that model is training well.
- Accuracy Subplot: The bottom subplot shows the training accuracy (blue line) and validation accuracy (orange line) throughout several epochs. Until it reaches a plateau, a well-trained model should show rising training accuracy. The validation accuracy illustrates how well the model functions with unknown data. High training accuracy and validation accuracy that is nearly equal to the training accuracy—a sign of strong generalization—are generally our goals.



Final result on test dataset (totally unseen):

- **accuracy for each class** is calculated by dividing the number of correctly classified images by the total number of images in that class. These accuracy values are then stored in the `percentages` dictionary with the class label as the key.

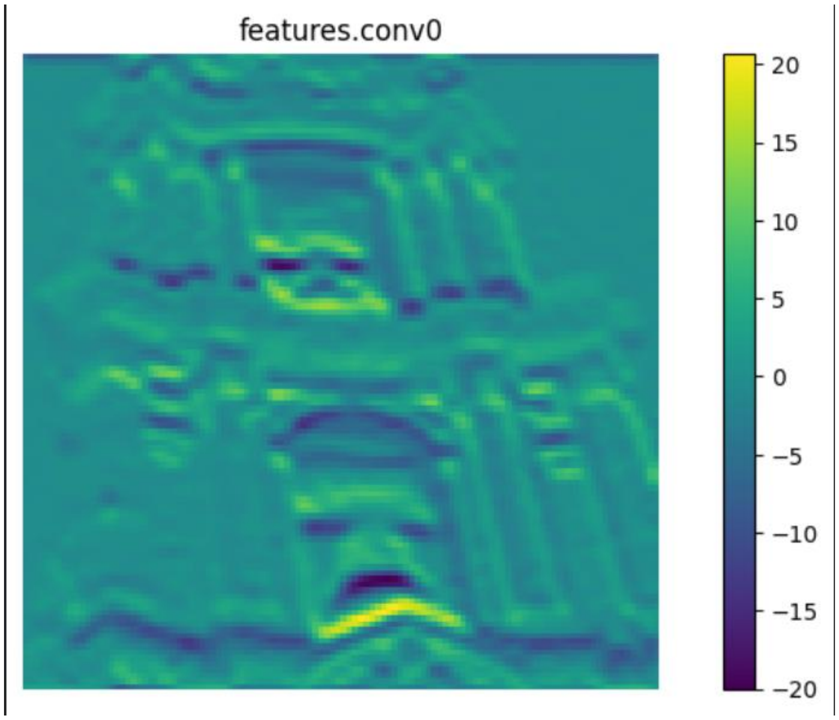


```
{0: 0.9278350515463918,
 3: 0.8616352201257862,
 4: 0.7735849056603774,
 6: 0.9056603773584906,
 2: 0.8615384615384616,
 9: 0.9411764705882353,
 7: 0.8944444444444445,
 5: 0.9649122807017544,
 8: 0.9532710280373832,
 1: 0.6585365853658537}
```

- **accuracy** on test dataset is 0.8926875593542261.

Feature Maps:

- this is feature map of first conv layer


features.conv0