CNN Model With PyTorch For Image Classification

Pranjal Soni · Follow
Published in TheCyPhy · 7 min read · Jan 9, 2021

446 2

Photo by Samer Khodeir on Unsplash

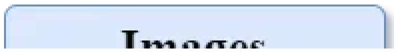
In this article, we discuss building a simple convolutional neural network(CNN) with PyTorch to classify images into different classes. By the end of this article, you become familiar with PyTorch, CNNs, padding, stride, max pooling and you are able to build your own CNN model for image classification. The dataset we are going to use is Intel Image Classification dataset available on Kaggle.

So let's begin, here is an outline of what this article going to cover:

1. Preparing The Dataset
2. Splitting Data and Prepare Batches
3. Base Model For Image Classification
4. Convolution, Padding, Stride, Pooling
5. CNN Model For Classification
6. Hyperparameters, Model Training, And Evaluation

Preparing the Dataset :

For training our model, we need a dataset which has images and label attached to it. But generally, the dataset available for image classification consists of images stored in corresponding folders. For example, our dataset consist of 6 types of images and they stored in corresponding folders.



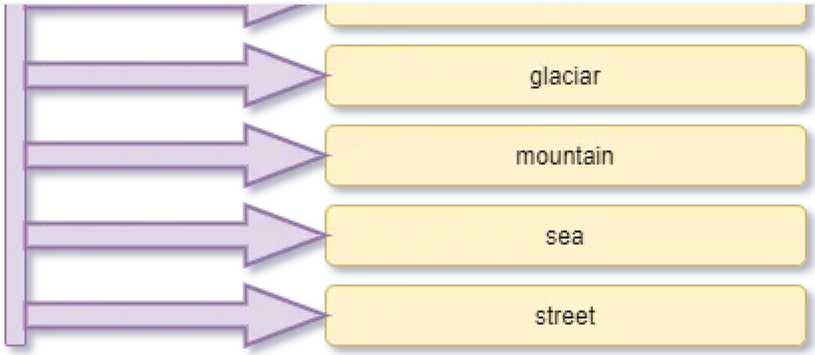


Diagram of the directory structure.

To prepare a dataset from such a structure, PyTorch provides *ImageFolder* class which makes the task easy for us to prepare the dataset. We simply have to pass the directory of our data to it and it provides the dataset which we can use to train the model.

```python
import torch
import torchvision
from torchvision import transforms
from torchvision.datasets import ImageFolder


#train and test data directory
data_dir = "../input/intel-image-classification/seg_train/seg_train/"
test_data_dir = "../input/intel-image-classification/seg_test/seg_test"


#load the train and test data
dataset = ImageFolder(data_dir,transform = transforms.Compose([
    transforms.Resize((150,150)),transforms.ToTensor()
]))
test_dataset = ImageFolder(test_data_dir,transforms.Compose([
    transforms.Resize((150,150)),transforms.ToTensor()
]))
```

data_loading hosted with ❤ by GitHub view raw

The *torchvision.transforms* module provides various functionality to preprocess the images, here first we resize the image for (150*150) shape and then transforms them into tensors.

```python
img, label = dataset[0]
print(img.shape,label)

#output :
#torch.Size([3, 150, 150]) 0
```

img_label.py hosted with ❤ by GitHub view raw

So our first image in the dataset has a shape (3,150,150) which means the image has 3 channels (RGB), height 150, and width 150. The image has a label 0, which represents the “buildings” class.

The image label set according to the class index in data.classes.

```python
print("Follwing classes are there : \n",dataset.classes)

#output:
#Follwing classes are there :
# ['buildings', 'forest', 'glacier', 'mountain', 'sea', 'street']
```

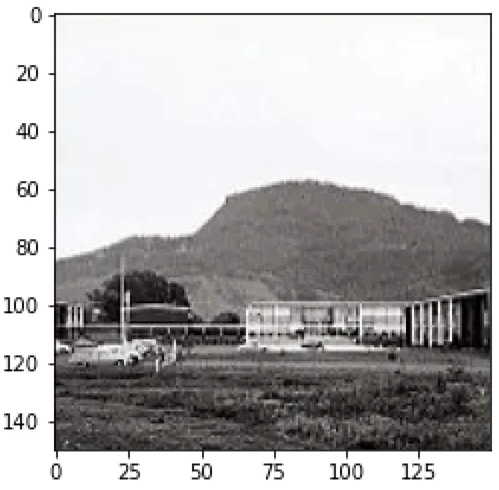
img_classes.py hosted with ❤ by GitHub view raw

So our dataset has 6 types of images in the dataset.

Exploring Images :

Our dataset consists of images in form of *Tensors,* imshow*()* method of matplotlib python library can be used to visualize images.

permute method reshapes the image from (3,150,150) to (150,150,3). The first image training data is of building as you can see below:



Splitting Data and Prepare Batches:

We can not pass the whole dataset into our model to train it, because our memory size is fixed and there is a high chance that our training data exceed the memory capacity of CPU or GPU, so we split the dataset into batches and instead of training the model on whole in a single phase. The batch size can be decided according to memory capacity, generally, it takes in power of 2. For example, the batch size can be 16, 32, 64, 128, 256, etc.

Here we take batches of size 128 and 2000 images from the data for validation and the rest of the data for training. To randomly split the images into training and testing, PyTorch provides random_split().

The data is divided into batches using the PyTorch *DataLoader* class. We create two objects train_dl and val_dl for training and validation data respectively by giving parameters training data and batch size into the *DataLoader* Class.

```python
from torch.utils.data.dataloader import DataLoader
from torch.utils.data import random_split

batch_size = 128
val_size = 2000
train_size = len(dataset) - val_size

train_data,val_data = random_split(dataset,[train_size,val_size])
print(f"Length of Train Data : {len(train_data)}")
print(f"Length of Validation Data : {len(val_data)}")

#output
#Length of Train Data : 12034
#Length of Validation Data : 2000

#load the train and validation into batches.
train_dl = DataLoader(train_data, batch_size, shuffle = True, num_workers = 4, pin_memor
val_dl = DataLoader(val_data, batch_size*2, num_workers = 4, pin_memory = True)
```

train_val_split.py hosted with ❤ by GitHub view raw

Visualizing the images:

To visualize images of a single batch, make_grid() can be used from torchvision utilities. It gives us an overall view of images in batch in the form of an image grid.

```python
from torchvision.utils import make_grid
import matplotlib.pyplot as plt

def show_batch(dl):
    """Plot images grid of single batch"""
    for images, labels in dl:
        fig,ax = plt.subplots(figsize = (16,12))
        ax.set_xticks([])
        ax.set_yticks([])
        ax.imshow(make_grid(images,nrow=16).permute(1,2,0))
        break

show_batch(train_dl)
```

make_grid.py hosted with ❤ by GitHub view raw

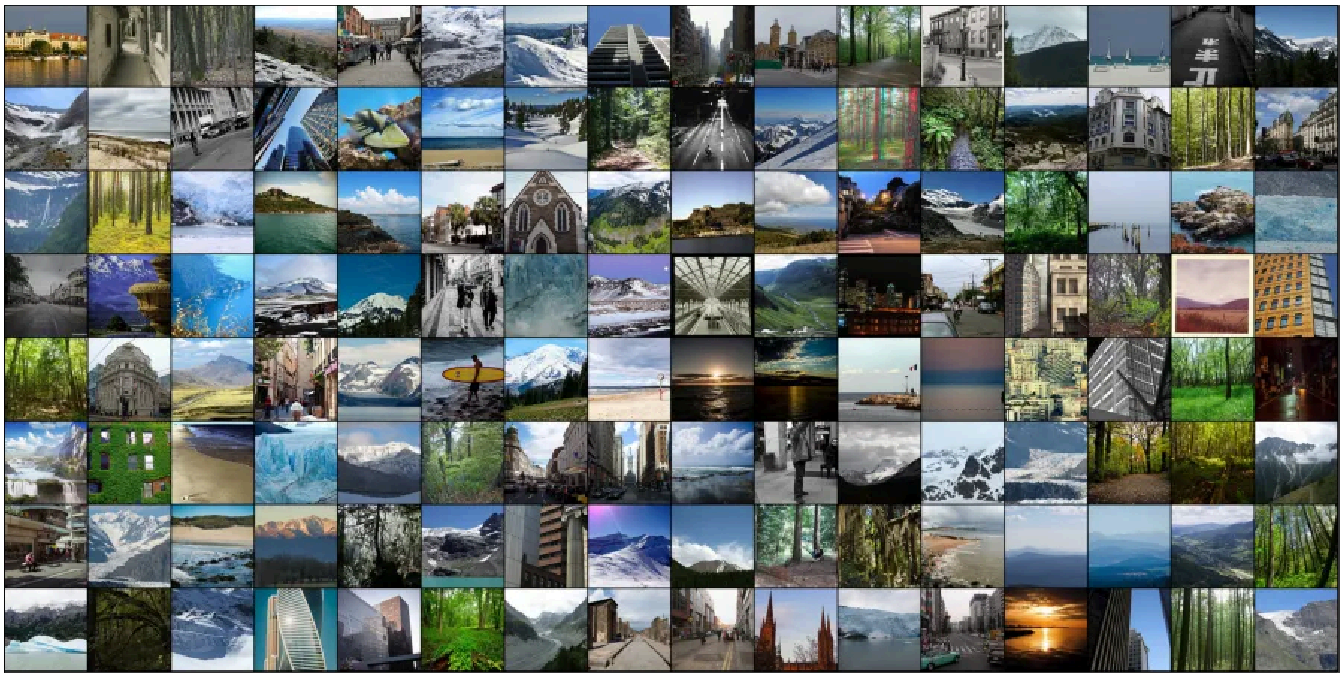


Image of the first batch

Base Model For Image Classification:

First, we prepare a base class that extends the functionality of torch.nn.Module (base class used to develop all neural networks). We add various functionalities to the base to train the model, validate the model, and get the result for each epoch. This is reusable and can be used for any image classification model, no need to rewrite this every time.

```python
import torch.nn as nn
import torch.nn.functional as F

class ImageClassificationBase(nn.Module):

    def training_step(self, batch):
        images, labels = batch
        out = self(images)                  # Generate predictions
        loss = F.cross_entropy(out, labels) # Calculate loss
        return loss

    def validation_step(self, batch):
        images, labels = batch
        out = self(images)                    # Generate predictions
        loss = F.cross_entropy(out, labels)   # Calculate loss
        acc = accuracy(out, labels)           # Calculate accuracy
        return {'val_loss': loss.detach(), 'val_acc': acc}

    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()   # Combine losses
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_accs).mean()      # Combine accuracies
        return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], train_loss: {:.4f}, val_loss: {:.4f}, val_acc: {:.4f}".format
            epoch, result['train_loss'], result['val_loss'], result['val_acc']))
```

image_claasfication_base.py hosted with ❤ by **GitHub** view raw

Convolution, Padding, Stride, Maxpooling;

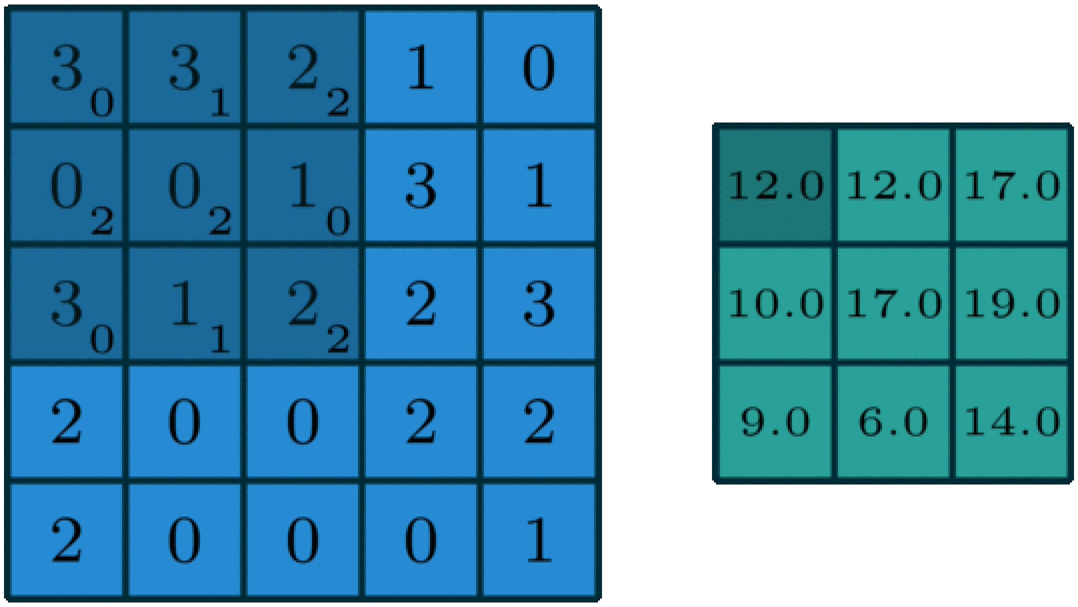
Now understand the concept of convolution, padding, and max-pooling that help our neural network to learn the features from the images.

Convolution :

> *According to Wikipedia Convolution is a mathematical operation which can be performed on two functions to generate third function that shows how shape of*

> *one modified by other.*

Here, the first function is the image tensor, and the second function is the matrix or tensor of the image with the same number of channels as our image called the kernel. Kernels are applied to the images to learn features from the images. Basically, the kernel performs dot product for each segment of the image and then sums the result and gives the output tensor. It can be understood easily by the following image:



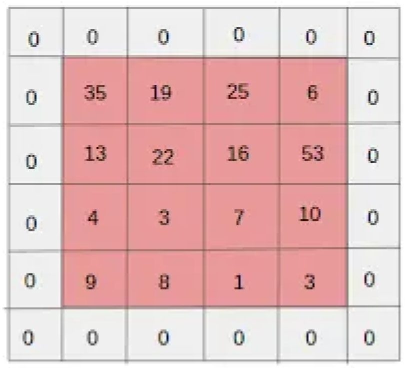
Source: https://miro.medium.com/max/1003/1*Zx-ZMLKab7VOCQTxdZ1OAw.gif

Padding:

When we apply a kernel to the image tensor in convolution, it reduces the size of the output tensor for the image. It causes two problems, first, it shrinks the output and the second is that pixel on the corner of the image losses its importance.

To resolve these issues we increase the shape of the image by adding some extra pixels to the border of the image tensor. It helps to increase the size of the image, and the pixel value of boundaries of images shifted inside the tensor. The features learned from them conveyed to further layers in deep neural networks.

In the image below zero-padding added to the 2-D tensor.



0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

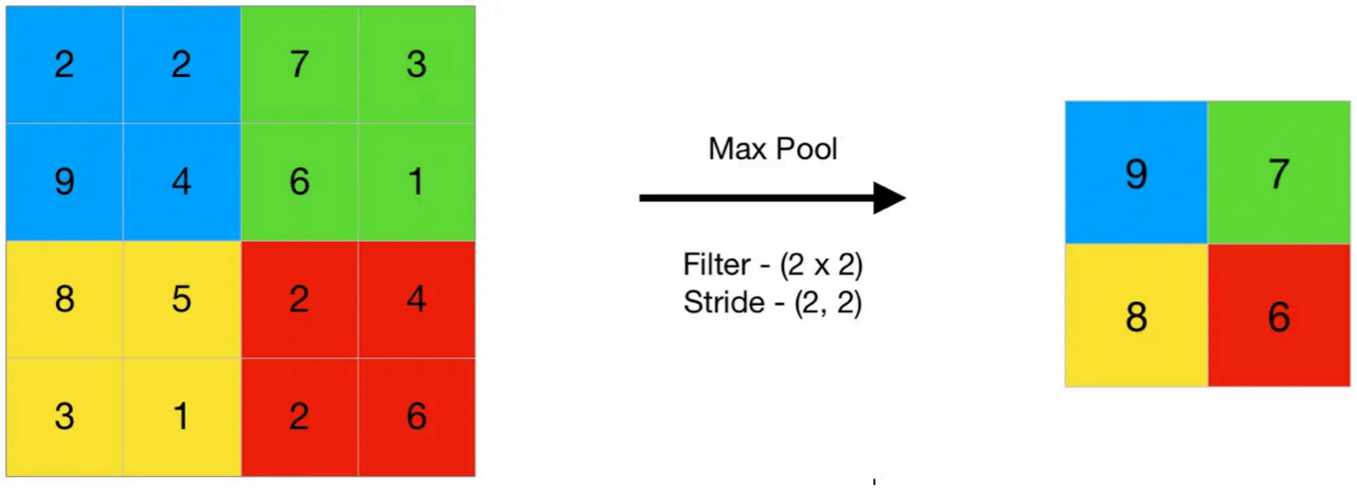
Source: https://xrds.acm.org/blog/wp-content/uploads/2016/06/Figure_3.png

Stride:

Stride controls the activity of the kernel, how the kernel moves across the image. For example, if the stride is set to (1,1), the kernel moves across the width and height by 1 pixel at a time. The first kernel moves across the width by 1 pixel, after completing the operation across the width it moves 1 pixel in height and again repeats the process. If the stride set (2,2) then the kernel moves across the image tensor by two pixels.

Pooling:

The pooling layer helps to summarise the result obtained by the convolution layer (also called feature map) in a lower dimension. There are various types of pooling like Max-Pooling, Average-Pooling, etc. Max-pooling often used, the image below described it more precisely:



Source

CNN Model For Classification:

After knowing all these concepts now we define our CNN model, which includes all these concepts to learn the features from the images and train the model.

In this model, there are 3 CNN blocks, and each block consists of 2 convolution layers and 1 max-pooling layer. Relu activation function is used to remove negative values from the feature map because there can not be negative values for any pixel value. Stride(1,1) used and padding is also 1.

After applying convolution and extract features from the image, a flatten layer is used to flat the tensor which has 3 dimensions. The flatten layer converts the tensor to one-dimensional. Then 3 linear added to reduce the size of the tensor and learn the features.

Architecture of our CNN Model:

```python
class NaturalSceneClassification(ImageClassificationBase):
    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(

            nn.Conv2d(3, 32, kernel_size = 3, padding = 1),
            nn.ReLU(),
            nn.Conv2d(32,64, kernel_size = 3, stride = 1, padding = 1),
            nn.ReLU(),
            nn.MaxPool2d(2,2),

            nn.Conv2d(64, 128, kernel_size = 3, stride = 1, padding = 1),
            nn.ReLU(),
            nn.Conv2d(128 ,128, kernel_size = 3, stride = 1, padding = 1),
            nn.ReLU(),
            nn.MaxPool2d(2,2),

            nn.Conv2d(128, 256, kernel_size = 3, stride = 1, padding = 1),
            nn.ReLU(),
            nn.Conv2d(256,256, kernel_size = 3, stride = 1, padding = 1),
            nn.ReLU(),
            nn.MaxPool2d(2,2),

            nn.Flatten(),
            nn.Linear(82944,1024),
            nn.ReLU(),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Linear(512,6)
        )

    def forward(self, xb):
        return self.network(xb)
```

natural_scene.py hosted with ❤️ by **GitHub** view raw

Hyperparameters, Model Training, And Evaluation:

Now we have to train the natural scene classification model on the training dataset. So that first defines the fit, evaluation, and accuracy methods.

```python
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))


@torch.no_grad()
def evaluate(model, val_loader):
    model.eval()
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)


def fit(epochs, lr, model, train_loader, val_loader, opt_func = torch.optim.SGD):

    history = []
    optimizer = opt_func(model.parameters(),lr)
    for epoch in range(epochs):

        model.train()
        train_losses = []
        for batch in train_loader:
            loss = model.training_step(batch)
            train_losses.append(loss)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

        result = evaluate(model, val_loader)
        result['train_loss'] = torch.stack(train_losses).mean().item()
        model.epoch_end(epoch, result)
        history.append(result)

    return history
```

helper_functions.py hosted with ❤ by **GitHub** view raw

Now we train our model for the different hyperparameters to get the best fit for the model. Here I train the model for 30 epochs, and a learning rate 0.001 and get 80% accuracy for the test data.

```
num_epochs = 30
opt_func = torch.optim.Adam
lr = 0.001

#fitting the model on training data and record the result after each epoch
history = fit(num_epochs, lr, model, train_dl, val_dl, opt_func)
```

Results for each epoch :

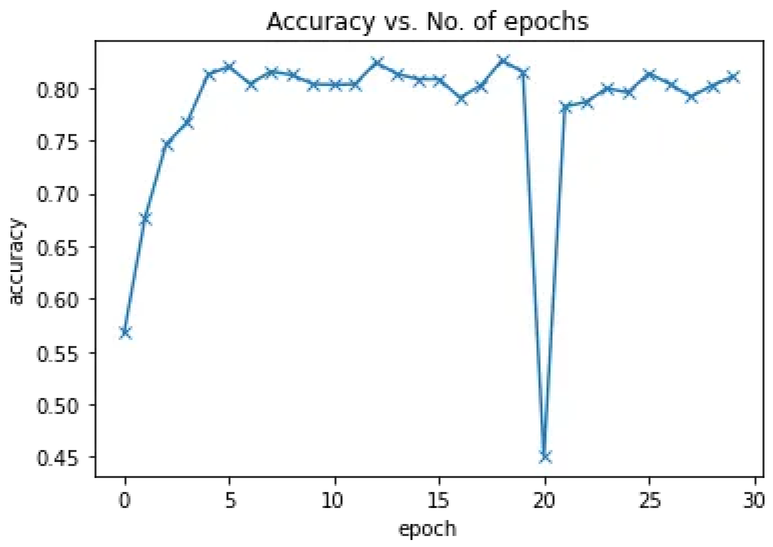
```
Epoch [0], train_loss: 1.2263, val_loss: 1.1289, val_acc: 0.5676
Epoch [1], train_loss: 0.8951, val_loss: 0.8003, val_acc: 0.6766
Epoch [2], train_loss: 0.7444, val_loss: 0.6793, val_acc: 0.7462
Epoch [3], train_loss: 0.5978, val_loss: 0.7027, val_acc: 0.7670
Epoch [4], train_loss: 0.5132, val_loss: 0.5253, val_acc: 0.8123
Epoch [5], train_loss: 0.4110, val_loss: 0.5356, val_acc: 0.8197
Epoch [6], train_loss: 0.3137, val_loss: 0.6205, val_acc: 0.8035
Epoch [7], train_loss: 0.2375, val_loss: 0.6337, val_acc: 0.8153
Epoch [8], train_loss: 0.1705, val_loss: 0.7263, val_acc: 0.8119
Epoch [9], train_loss: 0.1467, val_loss: 0.7694, val_acc: 0.8032
Epoch [10], train_loss: 0.1044, val_loss: 0.9002, val_acc: 0.8026
Epoch [11], train_loss: 0.1071, val_loss: 0.9872, val_acc: 0.8033
Epoch [12], train_loss: 0.0509, val_loss: 1.0953, val_acc: 0.8236
Epoch [13], train_loss: 0.0264, val_loss: 1.2699, val_acc: 0.8131
Epoch [14], train_loss: 0.0281, val_loss: 1.1232, val_acc: 0.8079
Epoch [15], train_loss: 0.0377, val_loss: 1.2976, val_acc: 0.8080
Epoch [16], train_loss: 0.0347, val_loss: 1.2475, val_acc: 0.7905
Epoch [17], train_loss: 0.0287, val_loss: 1.3546, val_acc: 0.8020
Epoch [18], train_loss: 0.0385, val_loss: 1.2569, val_acc: 0.8256
Epoch [19], train_loss: 0.0172, val_loss: 1.2729, val_acc: 0.8147
Epoch [20], train_loss: 0.0447, val_loss: 8.4484, val_acc: 0.4511
Epoch [21], train_loss: 0.7567, val_loss: 0.6766, val_acc: 0.7819
Epoch [22], train_loss: 0.1603, val_loss: 0.9974, val_acc: 0.7862
Epoch [23], train_loss: 0.0449, val_loss: 1.2573, val_acc: 0.7992
Epoch [24], train_loss: 0.0174, val_loss: 1.3928, val_acc: 0.7953
Epoch [25], train_loss: 0.0211, val_loss: 1.2213, val_acc: 0.8135
Epoch [26], train_loss: 0.0057, val_loss: 1.5535, val_acc: 0.8034
Epoch [27], train_loss: 0.0038, val_loss: 1.5412, val_acc: 0.7913
Epoch [28], train_loss: 0.0057, val_loss: 1.5192, val_acc: 0.8018
Epoch [29], train_loss: 0.0033, val_loss: 1.4707, val_acc: 0.8103
```

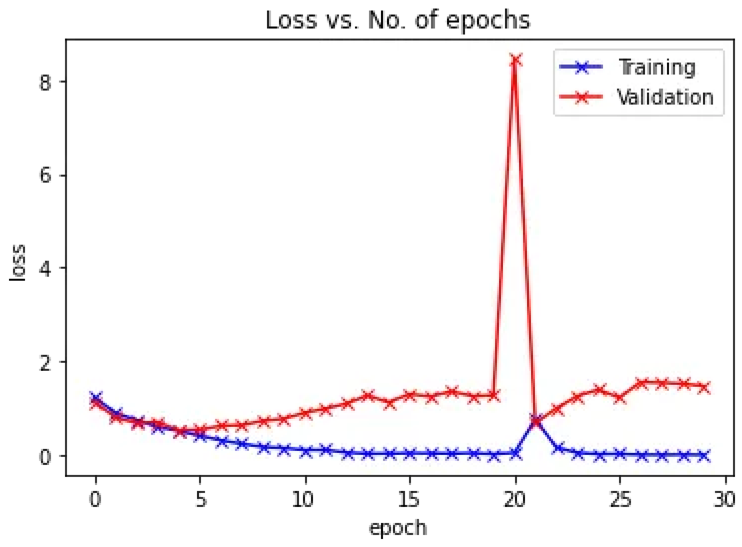
result hosted with ❤️ by **GitHub** view raw

Plotting the graph for accuracies and losses to visualize how the model improves its accuracy after each epoch:

```python
def plot_accuracies(history):
    """ Plot the history of accuracies"""
    accuracies = [x['val_acc'] for x in history]
    plt.plot(accuracies, '-x')
    plt.xlabel('epoch')
    plt.ylabel('accuracy')
    plt.title('Accuracy vs. No. of epochs');


plot_accuracies(history)

def plot_losses(history):
    """ Plot the losses in each epoch"""
    train_losses = [x.get('train_loss') for x in history]
    val_losses = [x['val_loss'] for x in history]
    plt.plot(train_losses, '-bx')
    plt.plot(val_losses, '-rx')
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.legend(['Training', 'Validation'])
    plt.title('Loss vs. No. of epochs');

plot_losses(history)
```

accuracy_and_loss.py hosted with ❤️ by **GitHub** view raw





To run the notebook on GPU and try out different hyperparameters or model you can check the following notebook on Kaggle:

natural-scene-classification

Explore and run machine learning code with Kaggle Notebooks | Using data from Intel Image Classification

www.kaggle.com

References: