AI VIETNAM
All-in-One Course
(TA Session)

Convolutional Neural Networks

Exercise

Dinh-Thang Duong – TA

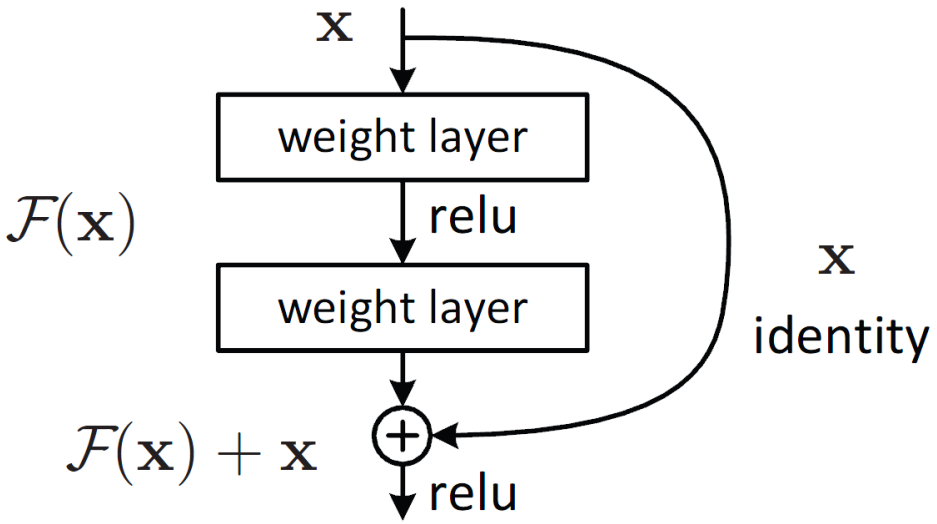
Year 2023

Outline

- ResNet
- DenseNet
- Question

ResNet

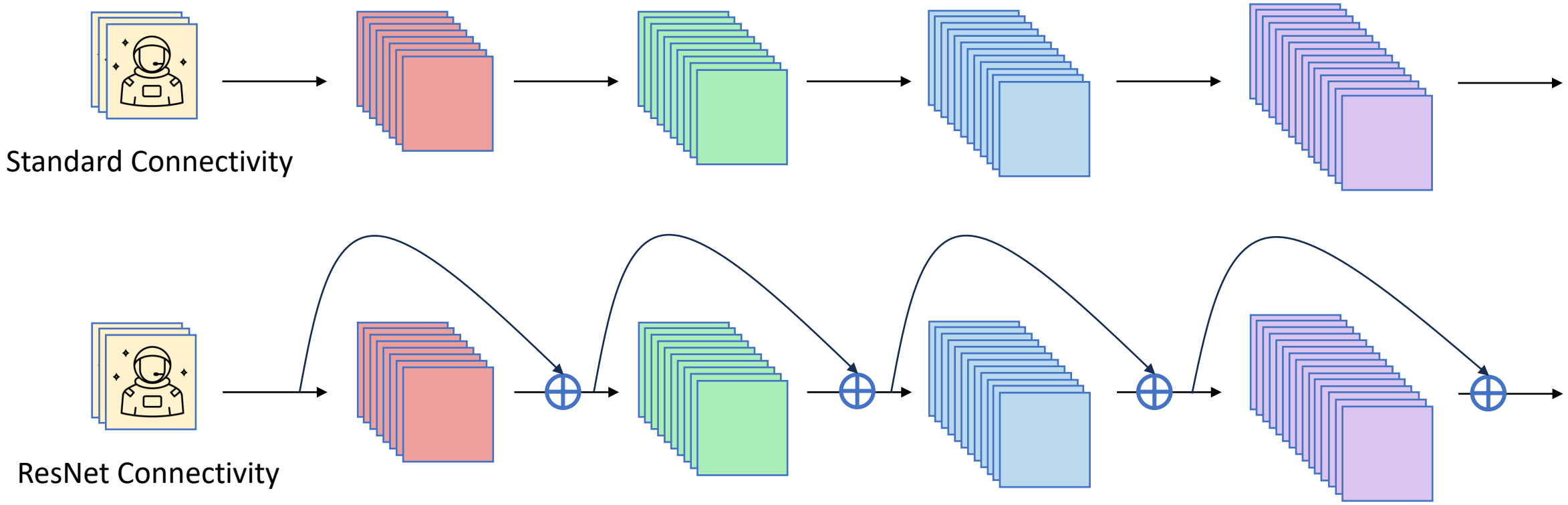
❖ ResNet Architecture



Residual Networks (ResNet): A type of CNNs first introduced in 2015. Its innovative approach involes using “residual blocks”, which allow the network to efficiently train extremely deep models by learning residual (difference) between the input and desired output.

ResNet

❖ ResNet Architecture



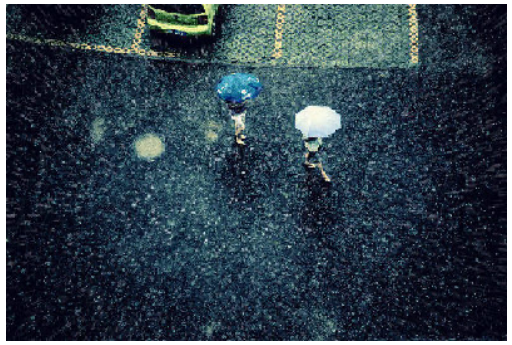
In **ResNet**, the output of a layer is combined with the input to create a residual connection or shorcut connection.

ResNet

❖ Coding Exercise

Problem Statement: Given the weather image dataset (download here). Train a model to determine which type of weather is in a image using ResNet.





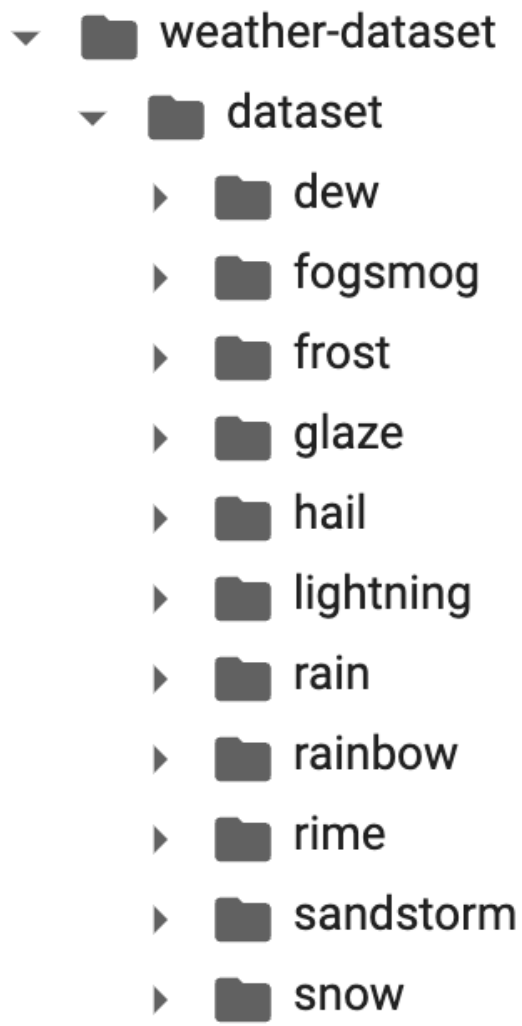

ResNet

❖ Step 1: Import libraries

```
import torch
import torch.nn as nn
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from PIL import Image
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
```

ResNet

❖ Step 2: Read dataset




```
1 root_dir = 'dataset/weather-dataset/dataset'
2 classes = {
3     label_idx: class_name \
4         for label_idx, class_name in enumerate(
5             sorted(os.listdir(root_dir))
6         )
7 }
```

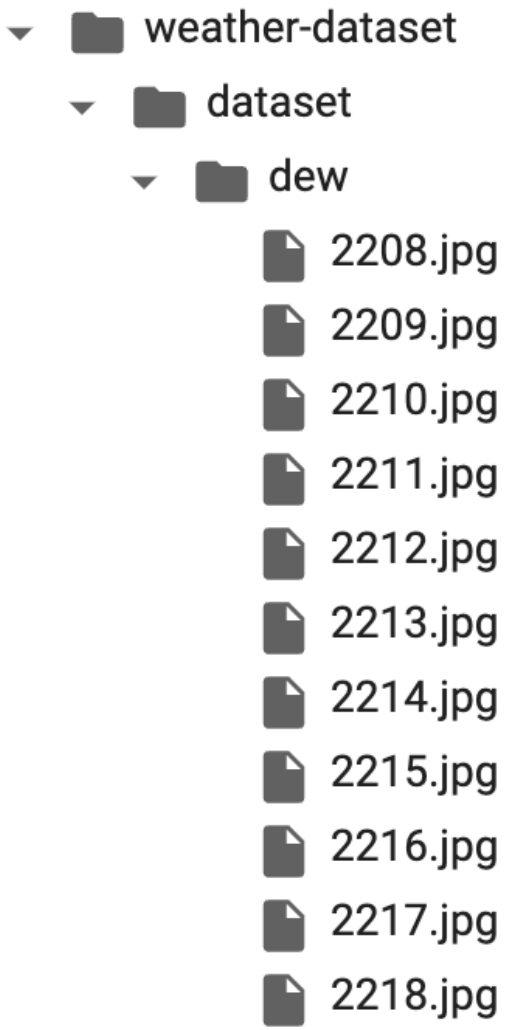
```
1 classes
```

```
{0: 'dew',
 1: 'fogsmog',
 2: 'frost',
 3: 'glaze',
 4: 'hail',
 5: 'lightning',
 6: 'rain',
 7: 'rainbow',
 8: 'rime',
 9: 'sandstorm',
 10: 'snow'}
```

Each folder name is also a class name, thus we could get list of classnames by reading over them.

ResNet

❖ Step 2: Read dataset




```
 9 img_paths = []
10 labels = []
11 for label_idx, class_name in classes.items():
12     class_dir = os.path.join(root_dir, class_name)
13     for img_filename in os.listdir(class_dir):
14         img_path = os.path.join(class_dir, img_filename)
15         img_paths.append(img_path)
16         labels.append(label_idx)
```

```
1 img_paths
```

```
['weather-dataset/dataset/dew/2405.jpg',
 'weather-dataset/dataset/dew/2340.jpg',
 'weather-dataset/dataset/dew/2772.jpg',
 'weather-dataset/dataset/dew/2794.jpg',
 'weather-dataset/dataset/dew/2827.jpg',
 'weather-dataset/dataset/dew/2235.jpg',
 'weather-dataset/dataset/dew/2396.jpg',
 'weather-dataset/dataset/dew/2349.jpg',
 'weather-dataset/dataset/dew/2879.jpg',
 'weather-dataset/dataset/dew/2734.jpg',
```

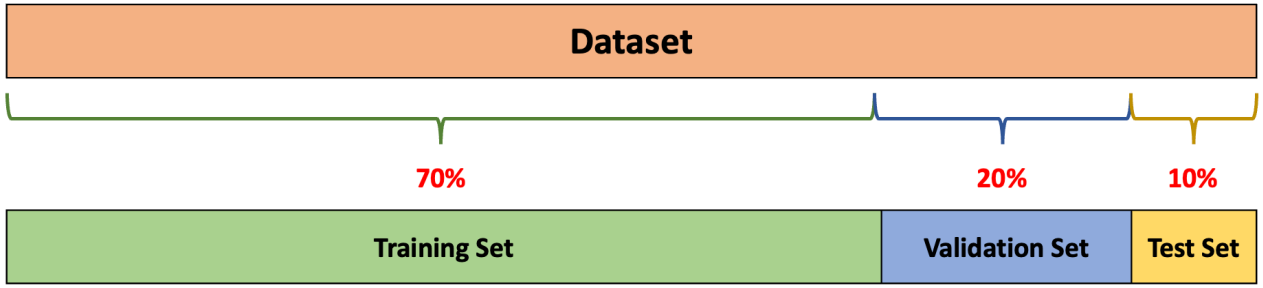
```
1 labels
```

```
[0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
```

We get all image paths as well as their corresponding label.

ResNet

❖ Step 3: Train, val, test split




```python
seed = 0
val_size = 0.2
test_size = 0.125
is_shuffle = True

X_train, X_val, y_train, y_val = train_test_split(
    img_paths, labels,
    test_size=val_size,
    random_state=seed,
    shuffle=is_shuffle
)

X_train, X_test, y_train, y_test = train_test_split(
    X_train, y_train,
    test_size=val_size,
    random_state=seed,
    shuffle=is_shuffle
)
```

ResNet

❖ Step 4: Create pytorch datasets

```python
class WeatherDataset(Dataset):
    def __init__(
        self,
        X, y,
        transform=None
    ):
        self.transform = transform
        self.img_paths = X
        self.labels = y

    def __len__(self):
        return len(self.img_paths)

    def __getitem__(self, idx):
        img_path = self.img_paths[idx]
        img = Image.open(img_path).convert("RGB")

        if self.transform:
            img = self.transform(img)

        return img, self.labels[idx]
```

```python
def transform(img, img_size=(224, 224)):
    img = img.resize(img_size)
    img = np.array(img)[..., :3]
    img = torch.tensor(img).permute(2, 0, 1).float()
    normalized_img = img / 255.0

    return normalized_img
```

```python
train_dataset = WeatherDataset(
    X_train, y_train,
    transform=transform
)
val_dataset = WeatherDataset(
    X_val, y_val,
    transform=transform
)
test_dataset = WeatherDataset(
    X_test, y_test,
    transform=transform
)
```

Declare train, val, test datasets object

ResNet

❖ Step 5: Create dataloader

```
train_batch_size = 512
test_batch_size = 8

train_loader = DataLoader(
    train_dataset,
    batch_size=train_batch_size,
    shuffle=True
)
val_loader = DataLoader(
    val_dataset,
    batch_size=test_batch_size,
    shuffle=False
)
test_loader = DataLoader(
    test_dataset,
    batch_size=test_batch_size,
    shuffle=False
)
```

We must also declare the value of batch size for training and testing.

DataLoader is an iterator, so we can get a sample (image_path, label) and visualize it.

```
train_features, train_labels = next(iter(train_loader))
print(f'Feature batch shape: {train_features.size()}')
print(f'Labels batch shape: {train_labels.size()}')
img = train_features[0].permute(1, 2, 0)
label = train_labels[0].item()
plt.imshow(img)
plt.axis('off')
plt.title(f'Label: {classes[label]}')
plt.show()
```

```
Feature batch shape: torch.Size([512, 3, 224, 224])
Labels batch shape: torch.Size([512])
```

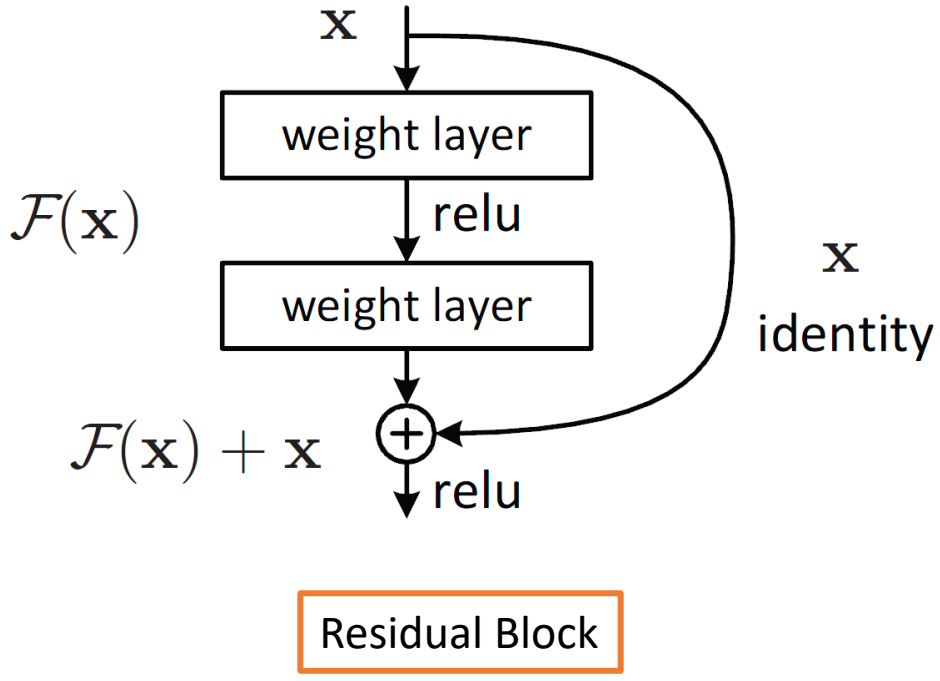
ResNet

❖ Step 6: ResNet Architecture

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

ResNet

❖ Step 6: Implement Residual Block

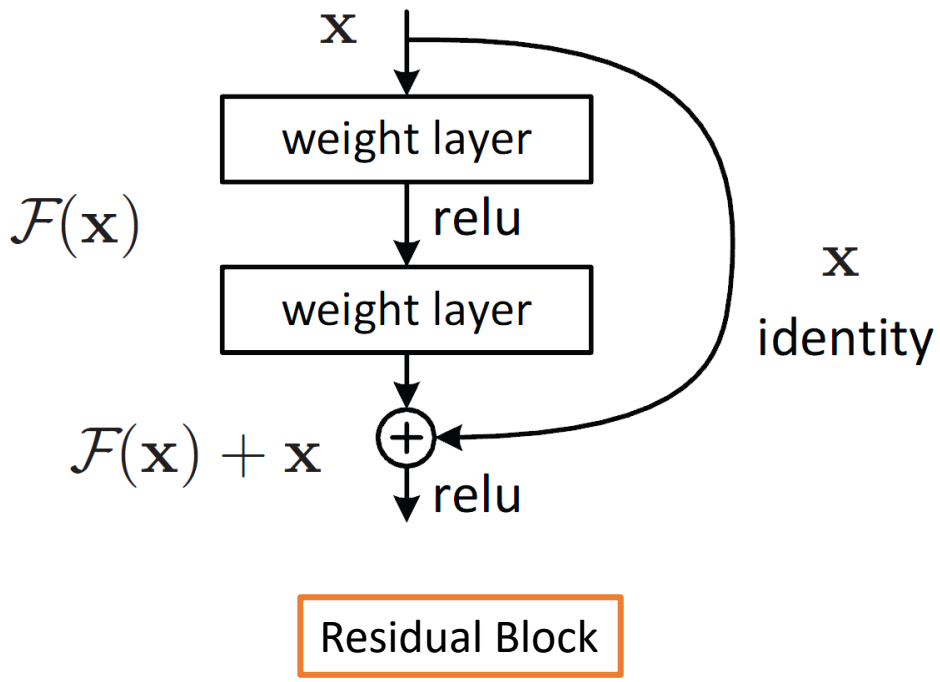


Residual Block

```python
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(
            in_channels, out_channels,
            kernel_size=3, stride=stride, padding=1
        )
        self.batch_norm1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(
            out_channels, out_channels,
            kernel_size=3, stride=1, padding=1
        )
        self.batch_norm2 = nn.BatchNorm2d(out_channels)

        self.downsample = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.downsample = nn.Sequential(
                nn.Conv2d(
                    in_channels, out_channels,
                    kernel_size=1, stride=stride
                ),
                nn.BatchNorm2d(out_channels)
            )
        self.relu = nn.ReLU()
```

ResNet

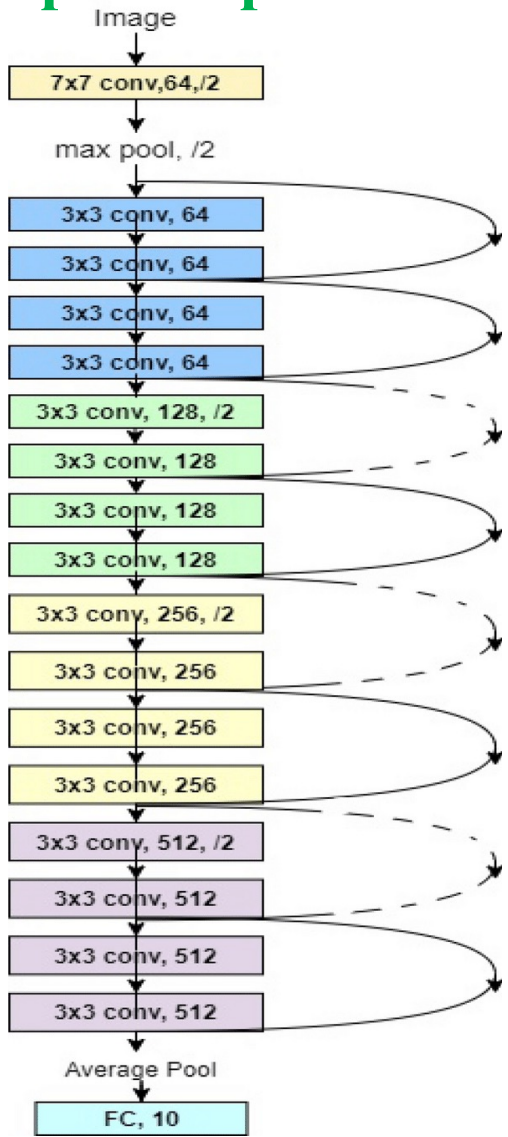
❖ Step 6: Implement Residual Block




```
26      def forward(self, x):
27          shortcut = x.clone()
28          x = self.conv1(x)
29          x = self.batch_norm1(x)
30          x = self.relu(x)
31          x = self.conv2(x)
32          x = self.batch_norm2(x)
33          x += self.downsample(shortcut)
34          x = self.relu(x)
35
36          return x
```

ResNet

❖ Step 7: Implement ResNet (ResNet-18)

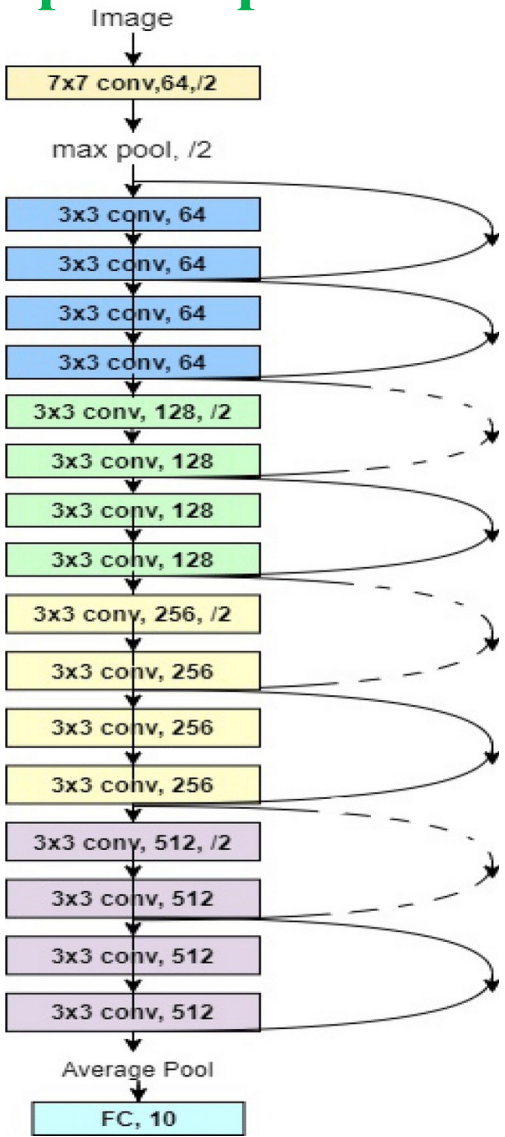



```python
    def create_layer(self, residual_block, in_channels, out_channels, n_blocks, stride):
        blocks = []
        first_block = residual_block(in_channels, out_channels, stride)
        blocks.append(first_block)

        for idx in range(1, n_blocks):
            block = residual_block(out_channels, out_channels, stride)
            blocks.append(block)

        block_sequential = nn.Sequential(*blocks)

        return block_sequential
```

```python
class ResNet(nn.Module):
    def __init__(self, residual_block, n_blocks_lst, n_classes):
        super(ResNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3)
        self.batch_norm1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.conv2 = self.create_layer(residual_block, 64, 64, n_blocks_lst[0], 1)
        self.conv3 = self.create_layer(residual_block, 64, 128, n_blocks_lst[1], 2)
        self.conv4 = self.create_layer(residual_block, 128, 256, n_blocks_lst[2], 2)
        self.conv5 = self.create_layer(residual_block, 256, 512, n_blocks_lst[3], 2)
        self.avgpool = nn.AdaptiveAvgPool2d(1)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(512, n_classes)
```

ResNet

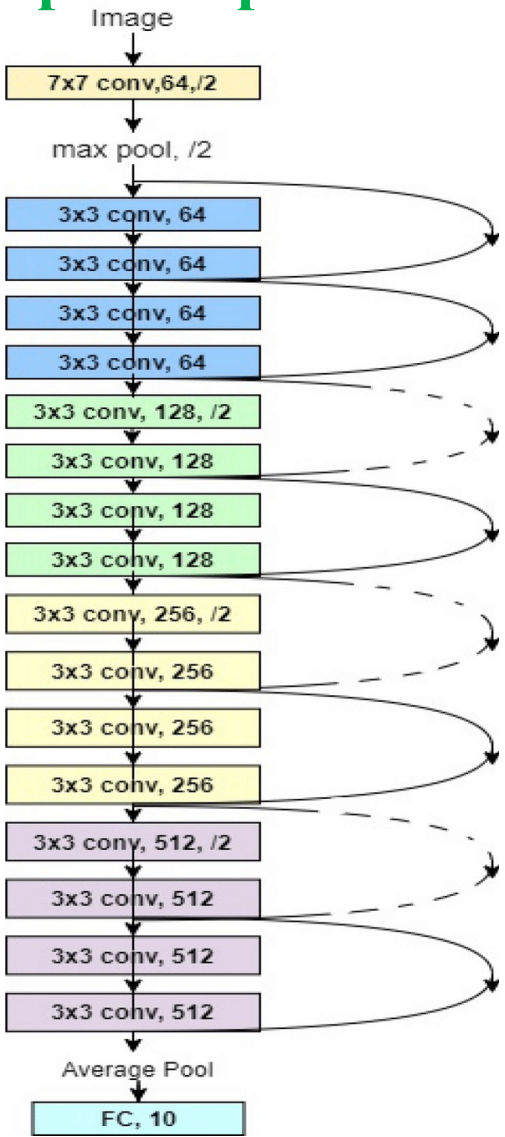
❖ Step 7: Implement ResNet (ResNet-18)




```python
class ResNet(nn.Module):
    def __init__(self, residual_block, n_blocks_lst, n_classes):
        super(ResNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3)
        self.batch_norm1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.conv2 = self.create_layer(residual_block, 64, 64, n_blocks_lst[0], 1)
        self.conv3 = self.create_layer(residual_block, 64, 128, n_blocks_lst[1], 2)
        self.conv4 = self.create_layer(residual_block, 128, 256, n_blocks_lst[2], 2)
        self.conv5 = self.create_layer(residual_block, 256, 512, n_blocks_lst[3], 2)
        self.avgpool = nn.AdaptiveAvgPool2d(1)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(512, n_classes)
```

ResNet

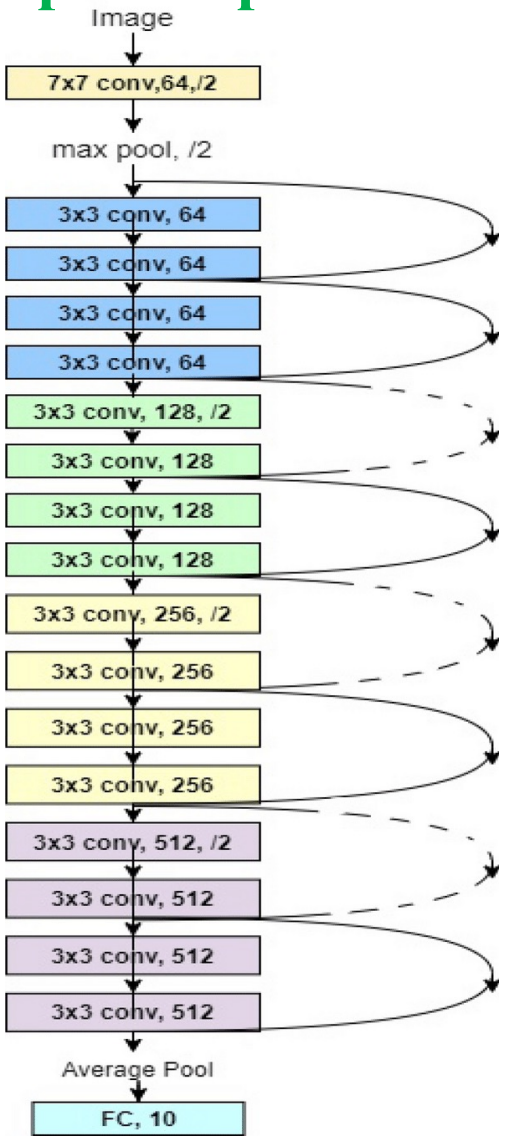
❖ Step 7: Implement ResNet (ResNet-18)




```
29      def forward(self, x):
30          x = self.conv1(x)
31          x = self.batch_norm1(x)
32          x = self.maxpool(x)
33          x = self.relu(x)
34          x = self.conv2(x)
35          x = self.conv3(x)
36          x = self.conv4(x)
37          x = self.conv5(x)
38          x = self.avgpool(x)
39          x = self.flatten(x)
40          x = self.fc1(x)
41
42          return x
```

ResNet

❖ Step 7: Implement ResNet (ResNet-18)




```python
n_classes = len(list(classes.keys()))
device = 'cuda' if torch.cuda.is_available() else 'cpu'

model = ResNet(
    residual_block=ResidualBlock,
    n_blocks_lst=[2, 2, 2, 2],
    n_classes=n_classes
).to(device)
```

```python
model.eval()

dummy_tensor = torch.randn(1, 3, 224, 224).to(device)

with torch.no_grad():
    output = model(dummy_tensor)

print('Output shape:', output.shape)
```

```
Output shape: torch.Size([1, 11])
```

After initialize the model, we should whether it works correctly or not

ResNet

❖ Step 8: Create fit function

From line 1 to line 9: Declare fit() function with the following parameters:

- model: the pytorch model to use.
- train_loader: the dataloader of train set.
- val_loader: the dataloader of val set.
- criterion: the pytorch loss function.
- optimizer: the pytorch learning algorithm.
- device: 'cuda' or 'cpu'.
- epochs: number of epochs.

```
1  def fit(
2      model,
3      train_loader,
4      val_loader,
5      criterion,
6      optimizer,
7      device,
8      epochs
9  ):
10     train_losses = []
11     val_losses = []
12
13     for epoch in range(epochs):
14         batch_train_losses = []
15
16         model.train()
17         for idx, (inputs, labels) in enumerate(train_loader):
18             inputs, labels = inputs.to(device), labels.to(device)
19
20             optimizer.zero_grad()
21             outputs = model(inputs)
22             loss = criterion(outputs, labels)
23             loss.backward()
24             optimizer.step()
25
26             batch_train_losses.append(loss.item())
27
28         train_loss = sum(batch_train_losses) / len(batch_train_losses)
29         train_losses.append(train_loss)
30
31         val_loss, val_acc = evaluate(
32             model, val_loader,
33             criterion, device
34         )
35         val_losses.append(val_loss)
36
37         print(f'EPOCH {epoch + 1}:\tTrain loss: {train_loss:.4f}\tVal loss: {val_loss:.4f}')
38
39     return train_losses, val_losses
```

ResNet

❖ Step 8: Create fit function

Line 10, 11: Declare two empty lists for storing losses on train set and val set over epochs. These will be the return values of our function.

```
1  def fit(
2      model,
3      train_loader,
4      val_loader,
5      criterion,
6      optimizer,
7      device,
8      epochs
9  ):
10     train_losses = []
11     val_losses = []
12
13     for epoch in range(epochs):
14         batch_train_losses = []
15
16         model.train()
17         for idx, (inputs, labels) in enumerate(train_loader):
18             inputs, labels = inputs.to(device), labels.to(device)
19
20             optimizer.zero_grad()
21             outputs = model(inputs)
22             loss = criterion(outputs, labels)
23             loss.backward()
24             optimizer.step()
25
26             batch_train_losses.append(loss.item())
27
28         train_loss = sum(batch_train_losses) / len(batch_train_losses)
29         train_losses.append(train_loss)
30
31         val_loss, val_acc = evaluate(
32             model, val_loader,
33             criterion, device
34         )
35         val_losses.append(val_loss)
36
37         print(f'EPOCH {epoch + 1}:\tTrain loss: {train_loss:.4f}\tVal loss: {val_loss:.4f}')
38
39     return train_losses, val_losses
```

ResNet

❖ Step 8: Create fit function

Line 13, 14, 15, 16: Iterate over each epoch, create an empty list for storing loss on each iteration and turn the training mode of the model on.

```
1  def fit(
2      model,
3      train_loader,
4      val_loader,
5      criterion,
6      optimizer,
7      device,
8      epochs
9  ):
10     train_losses = []
11     val_losses = []
12
13     for epoch in range(epochs):
14         batch_train_losses = []
15
16         model.train()
17         for idx, (inputs, labels) in enumerate(train_loader):
18             inputs, labels = inputs.to(device), labels.to(device)
19
20             optimizer.zero_grad()
21             outputs = model(inputs)
22             loss = criterion(outputs, labels)
23             loss.backward()
24             optimizer.step()
25
26             batch_train_losses.append(loss.item())
27
28         train_loss = sum(batch_train_losses) / len(batch_train_losses)
29         train_losses.append(train_loss)
30
31         val_loss, val_acc = evaluate(
32             model, val_loader,
33             criterion, device
34         )
35         val_losses.append(val_loss)
36
37         print(f'EPOCH {epoch + 1}:\tTrain loss: {train_loss:.4f}\tVal loss: {val_loss:.4f}')
38
39     return train_losses, val_losses
```

ResNet

❖ Step 8: Create fit function

Line 17, 18: Iterate over each batch of samples in train dataloader, convert the input tensor into correspoding device (CPU or GPU).

Line 20, 21: Clear the previous gradient and execute the prediction on inputs.

```
1  def fit(
2      model,
3      train_loader,
4      val_loader,
5      criterion,
6      optimizer,
7      device,
8      epochs
9  ):
10     train_losses = []
11     val_losses = []
12
13     for epoch in range(epochs):
14         batch_train_losses = []
15
16         model.train()
17         for idx, (inputs, labels) in enumerate(train_loader):
18             inputs, labels = inputs.to(device), labels.to(device)
19
20             optimizer.zero_grad()
21             outputs = model(inputs)
22             loss = criterion(outputs, labels)
23             loss.backward()
24             optimizer.step()
25
26             batch_train_losses.append(loss.item())
27
28         train_loss = sum(batch_train_losses) / len(batch_train_losses)
29         train_losses.append(train_loss)
30
31         val_loss, val_acc = evaluate(
32             model, val_loader,
33             criterion, device
34         )
35         val_losses.append(val_loss)
36
37         print(f'EPOCH {epoch + 1}:\tTrain loss: {train_loss:.4f}\tVal loss: {val_loss:.4f}')
38
39     return train_losses, val_losses
```

ResNet

❖ Step 8: Create fit function

Line 22, 23, 24, 26: Compute the loss, gradient and update the computed gradient to the optimizer. Then, append the computed loss to empty list declared on line 14.

```
1  def fit(
2      model,
3      train_loader,
4      val_loader,
5      criterion,
6      optimizer,
7      device,
8      epochs
9  ):
10     train_losses = []
11     val_losses = []
12
13     for epoch in range(epochs):
14         batch_train_losses = []
15
16         model.train()
17         for idx, (inputs, labels) in enumerate(train_loader):
18             inputs, labels = inputs.to(device), labels.to(device)
19
20             optimizer.zero_grad()
21             outputs = model(inputs)
22             loss = criterion(outputs, labels)
23             loss.backward()
24             optimizer.step()
25
26             batch_train_losses.append(loss.item())
27
28         train_loss = sum(batch_train_losses) / len(batch_train_losses)
29         train_losses.append(train_loss)
30
31         val_loss, val_acc = evaluate(
32             model, val_loader,
33             criterion, device
34         )
35         val_losses.append(val_loss)
36
37         print(f'EPOCH {epoch + 1}:\tTrain loss: {train_loss:.4f}\tVal loss: {val_loss:.4f}')
38
39     return train_losses, val_losses
```

ResNet

❖ Step 8: Create fit function

Line 28, 29: With a list of loss value of each iteration, we average them to get a single representative value for the current epoch.

```python
def fit(
    model,
    train_loader,
    val_loader,
    criterion,
    optimizer,
    device,
    epochs
):
    train_losses = []
    val_losses = []

    for epoch in range(epochs):
        batch_train_losses = []

        model.train()
        for idx, (inputs, labels) in enumerate(train_loader):
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            batch_train_losses.append(loss.item())

        train_loss = sum(batch_train_losses) / len(batch_train_losses)
        train_losses.append(train_loss)

        val_loss, val_acc = evaluate(
            model, val_loader,
            criterion, device
        )
        val_losses.append(val_loss)

        print(f'EPOCH {epoch + 1}:\tTrain loss: {train_loss:.4f}\tVal loss: {val_loss:.4f}')

    return train_losses, val_losses
```

ResNet

❖ Step 8: Create fit function

From line 31 to 39: Evaluate the model on val set, append the val loss to empty list declared in line 11, print the training result on the current epoch. If the iteration over epoch ends, return the train_losses and val_losses.

```
1  def fit(
2      model,
3      train_loader,
4      val_loader,
5      criterion,
6      optimizer,
7      device,
8      epochs
9  ):
10     train_losses = []
11     val_losses = []
12
13     for epoch in range(epochs):
14         batch_train_losses = []
15
16         model.train()
17         for idx, (inputs, labels) in enumerate(train_loader):
18             inputs, labels = inputs.to(device), labels.to(device)
19
20             optimizer.zero_grad()
21             outputs = model(inputs)
22             loss = criterion(outputs, labels)
23             loss.backward()
24             optimizer.step()
25
26             batch_train_losses.append(loss.item())
27
28         train_loss = sum(batch_train_losses) / len(batch_train_losses)
29         train_losses.append(train_loss)
30
31         val_loss, val_acc = evaluate(
32             model, val_loader,
33             criterion, device
34         )
35         val_losses.append(val_loss)
36
37         print(f'EPOCH {epoch + 1}:\tTrain loss: {train_loss:.4f}\tVal loss: {val_loss:.4f}')
38
39     return train_losses, val_losses
```

ResNet

❖ Step 9: Training

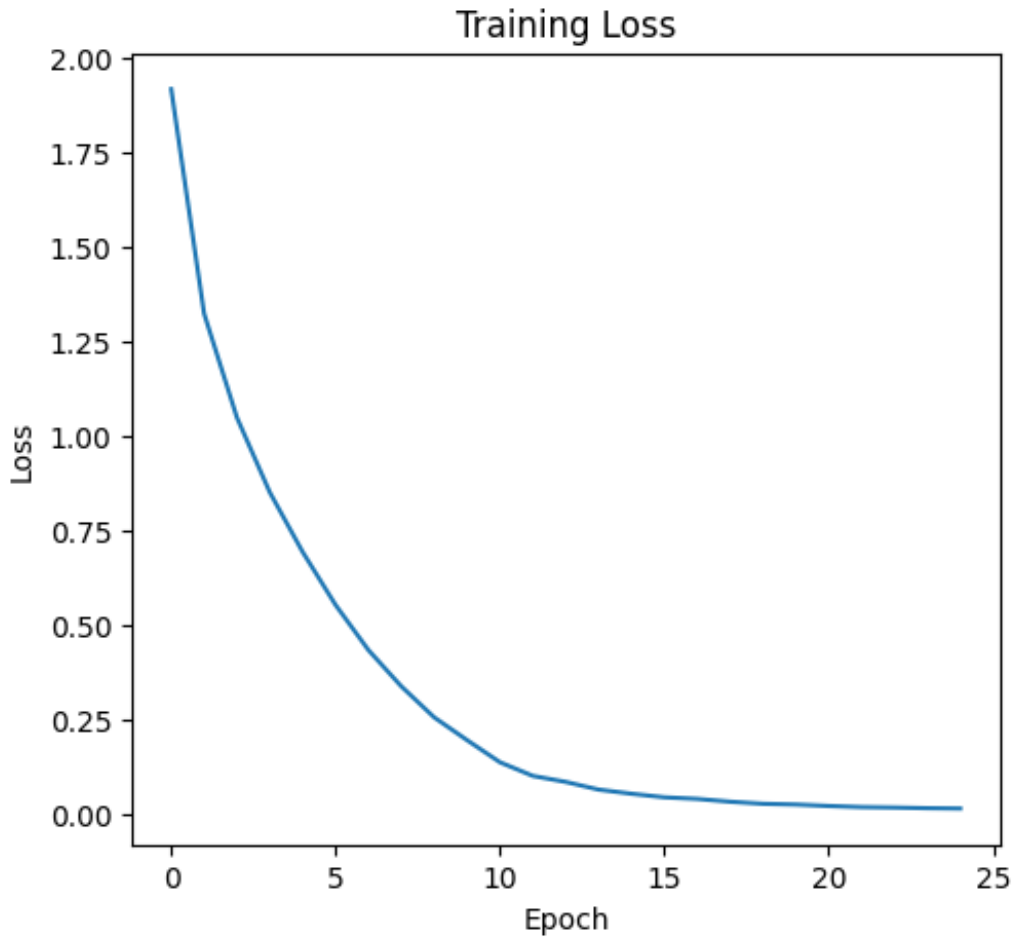
```
lr = 1e-2
epochs = 25

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(
    model.parameters(),
    lr=lr
)
```

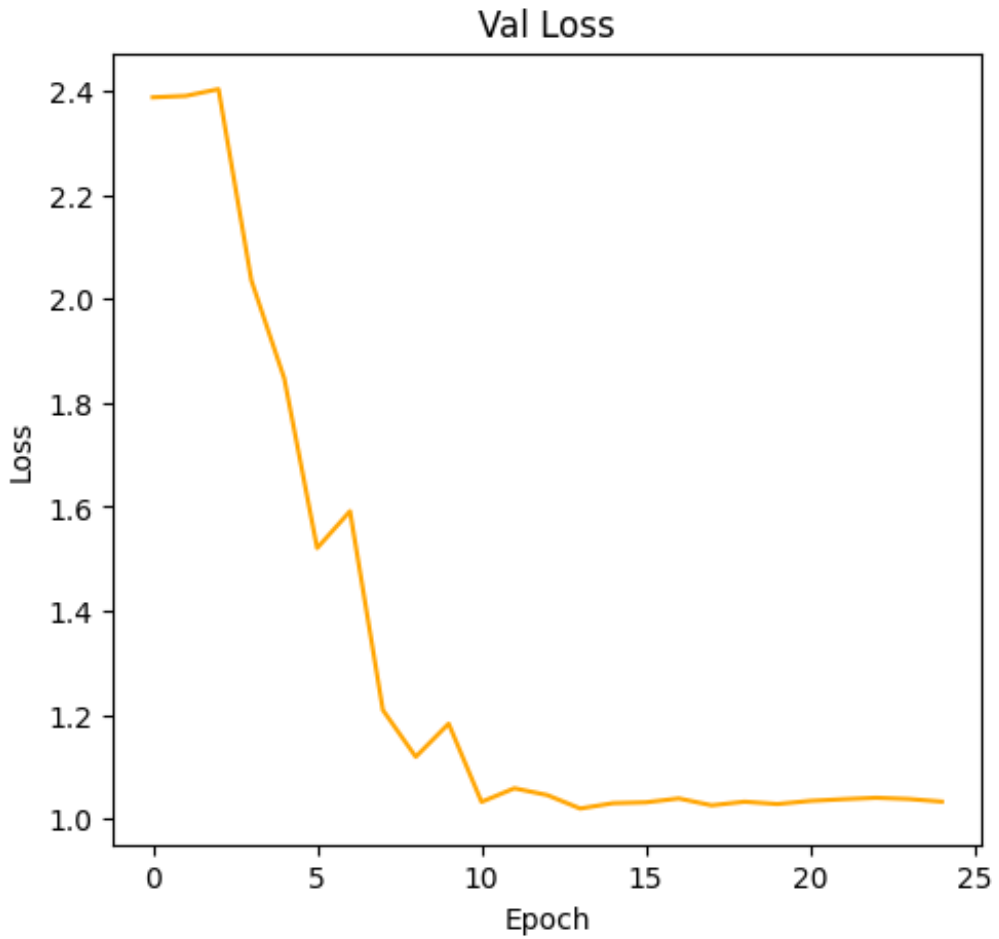
```
train_losses, val_losses = fit(
    model,
    train_loader,
    val_loader,
    criterion,
    optimizer,
    device,
    epochs
)
```

```
EPOCH 1:        Train loss: 1.9155      Val loss: 2.3871
EPOCH 2:        Train loss: 1.3232      Val loss: 2.3897
EPOCH 3:        Train loss: 1.0485      Val loss: 2.4028
EPOCH 4:        Train loss: 0.8511      Val loss: 2.0351
EPOCH 5:        Train loss: 0.6936      Val loss: 1.8467
EPOCH 6:        Train loss: 0.5541      Val loss: 1.5207
EPOCH 7:        Train loss: 0.4349      Val loss: 1.5914
EPOCH 8:        Train loss: 0.3392      Val loss: 1.2092
EPOCH 9:        Train loss: 0.2574      Val loss: 1.1194
EPOCH 10:       Train loss: 0.1973      Val loss: 1.1830
EPOCH 11:       Train loss: 0.1389      Val loss: 1.0327
EPOCH 12:       Train loss: 0.1029      Val loss: 1.0589
EPOCH 13:       Train loss: 0.0871      Val loss: 1.0458
EPOCH 14:       Train loss: 0.0667      Val loss: 1.0200
EPOCH 15:       Train loss: 0.0558      Val loss: 1.0302
EPOCH 16:       Train loss: 0.0464      Val loss: 1.0318
EPOCH 17:       Train loss: 0.0422      Val loss: 1.0393
EPOCH 18:       Train loss: 0.0348      Val loss: 1.0261
EPOCH 19:       Train loss: 0.0294      Val loss: 1.0330
EPOCH 20:       Train loss: 0.0274      Val loss: 1.0285
EPOCH 21:       Train loss: 0.0236      Val loss: 1.0346
EPOCH 22:       Train loss: 0.0206      Val loss: 1.0378
EPOCH 23:       Train loss: 0.0194      Val loss: 1.0406
EPOCH 24:       Train loss: 0.0178      Val loss: 1.0382
EPOCH 25:       Train loss: 0.0169      Val loss: 1.0332
```

ResNet

❖ Step 9: Training





ResNet

❖ Step 10: Create evaluation function

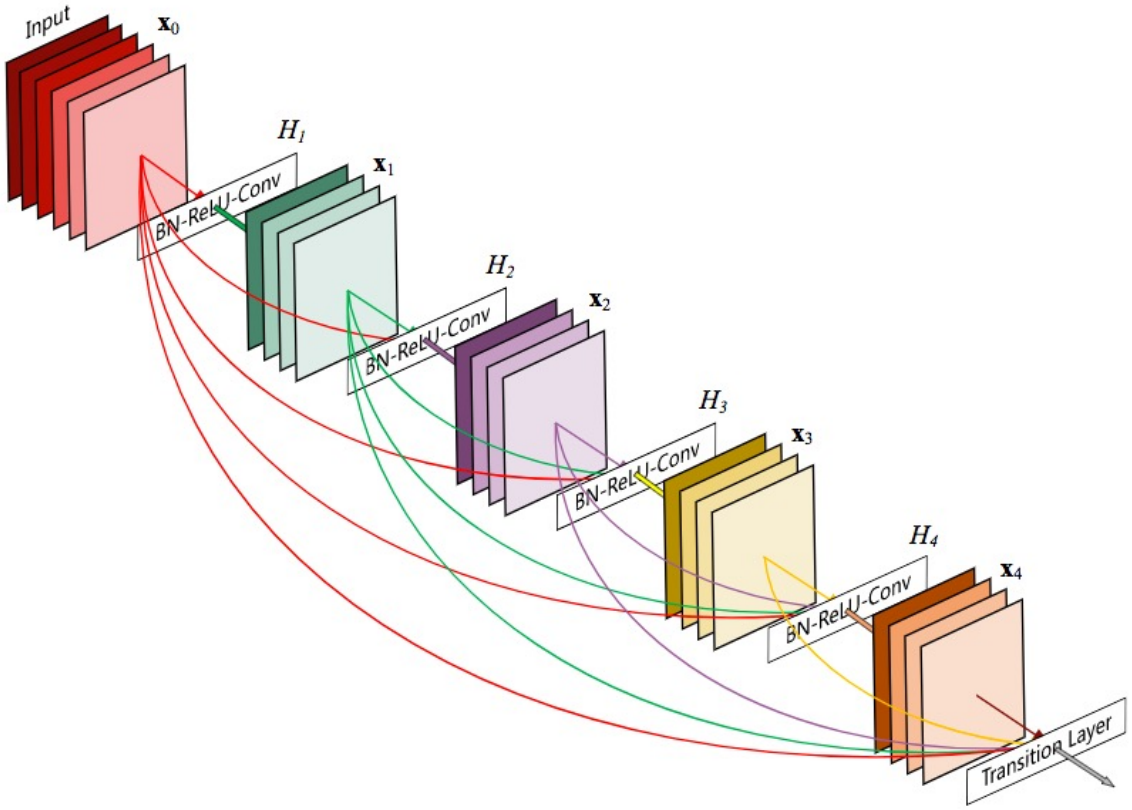
```python
def evaluate(model, dataloader, criterion, device):
    model.eval()
    correct = 0
    total = 0
    losses = []
    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            losses.append(loss.item())
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    loss = sum(losses) / len(losses)
    acc = correct / total

    return loss, acc
```

```python
val_loss, val_acc = evaluate(
    model,
    val_loader,
    criterion,
    device
)
test_loss, test_acc = evaluate(
    model,
    test_loader,
    criterion,
    device
)

print('Evaluation on val/test dataset')
print('Val accuracy: ', val_acc)
print('Test accuracy: ', test_acc)
```

```
Evaluation on val/test dataset
Val accuracy:  0.6758922068463219
Test accuracy:  0.70400728597449 91
```

DenseNet

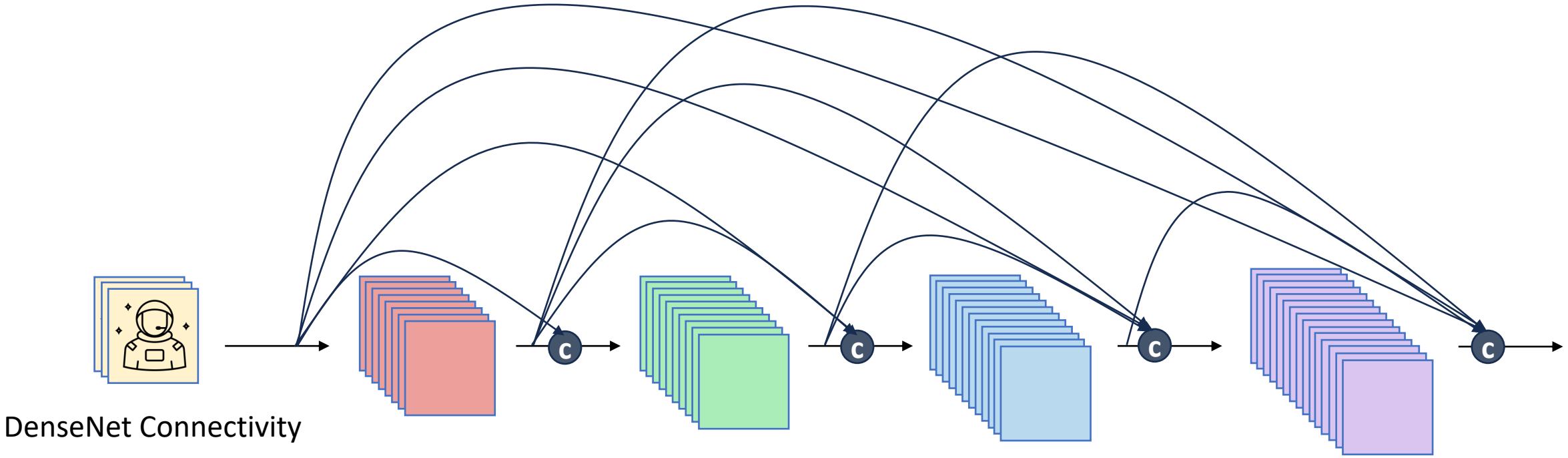
❖ DenseNet Architecture



Densely Connected Convolutional Networks (DenseNet): A type of CNNs introduced in 2017. It revolutionized CNNs by introducing dense connectivity (DenseBlock), facilitating information flow and feature reuse.

DenseNet

❖ DenseNet architecture



DenseNet Connectivity

In **DenseNet**, each layer is not only connected to the previous layer but also to all subsequent layers in the network.

DenseNet

❖ Coding Exercise

Problem Statement: Given the scenes image dataset (download here), build a scenes image classifier using DenseNet architecture.

DenseNet

❖ Step 1: Import libraries

```
1 import torch
2 import torch.nn as nn
3 import os
4 import numpy as np
5 import pandas as pd
6 import matplotlib.pyplot as plt
7
8 from PIL import Image
9 from torch.utils.data import Dataset, DataLoader
10 from sklearn.model_selection import train_test_split
```

DenseNet

❖ Step 2: Read dataset


```
root_dir = 'dataset/scenes_classification'
train_dir = os.path.join(root_dir, 'train')
test_dir = os.path.join(root_dir, 'val')

classes = {
    label_idx: class_name \
        for label_idx, class_name in enumerate(
            sorted(os.listdir(train_dir))
        )
}
```

```
classes
```

```
{0: 'buildings',
 1: 'forest',
 2: 'glacier',
 3: 'mountain',
 4: 'sea',
 5: 'street'}
```

Each folder name is also a class name, thus we could get list of classnames by reading over them.

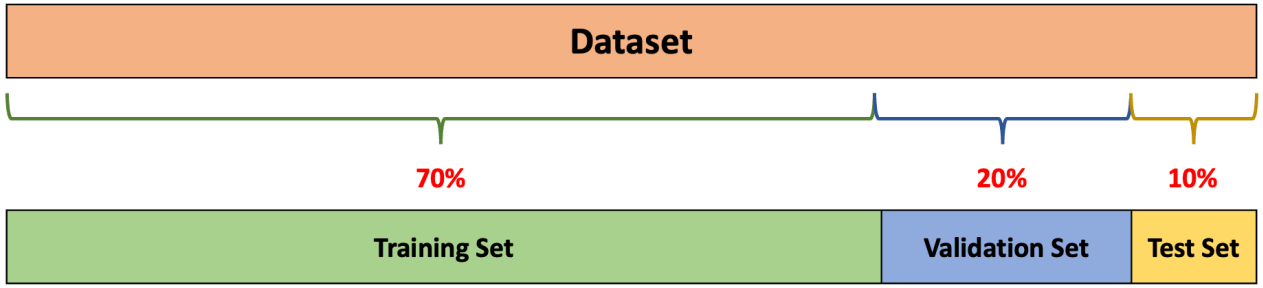
DenseNet

❖ Step 2: Read dataset


```
X_train = []
y_train = []
X_test = []
y_test = []
for dataset_path in [train_dir, test_dir]:
    for label_idx, class_name in classes.items():
        class_dir = os.path.join(dataset_path, class_name)
        for img_filename in os.listdir(class_dir):
            img_path = os.path.join(class_dir, img_filename)
            if 'train' in dataset_path:
                X_train.append(img_path)
                y_train.append(label_idx)
            else:
                X_test.append(img_path)
                y_test.append(label_idx)
```

DenseNet

❖ Step 3: Train, val, test split




```
seed = 0
val_size = 0.2
is_shuffle = True

X_train, X_val, y_train, y_val = train_test_split(
    X_train, y_train,
    test_size=val_size,
    random_state=seed,
    shuffle=is_shuffle
)
```

We only need to split once since we already have test set

DenseNet

❖ Step 4: Create pytorch dataset

```python
class ScenesDataset(Dataset):
    def __init__(
        self,
        X, y,
        transform=None
    ):
        self.transform = transform
        self.img_paths = X
        self.labels = y

    def __len__(self):
        return len(self.img_paths)

    def __getitem__(self, idx):
        img_path = self.img_paths[idx]
        img = Image.open(img_path).convert("RGB")

        if self.transform:
            img = self.transform(img)

        return img, self.labels[idx]
```

```python
def transform(img, img_size=(224, 224)):
    img = img.resize(img_size)
    img = np.array(img)[..., :3]
    img = torch.tensor(img).permute(2, 0, 1).float()
    normalized_img = img / 255.0

    return normalized_img
```

```python
train_dataset = ScenesDataset(
    X_train, y_train,
    transform=transform
)
val_dataset = ScenesDataset(
    X_val, y_val,
    transform=transform
)
test_dataset = ScenesDataset(
    X_test, y_test,
    transform=transform
)
```

Declare train, val, test datasets object

DenseNet

❖ Step 5: Create dataloader

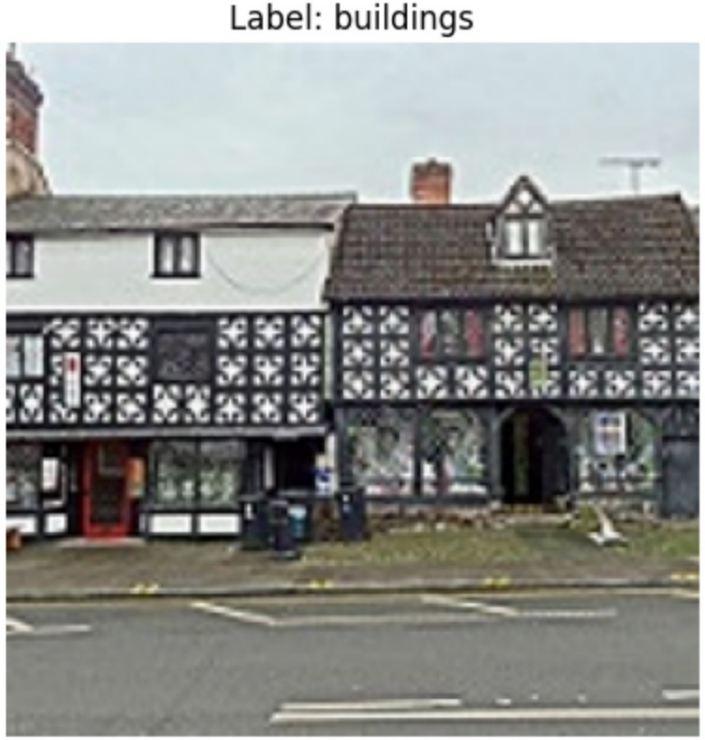
```python
train_batch_size = 64
test_batch_size = 8

train_loader = DataLoader(
    train_dataset,
    batch_size=train_batch_size,
    shuffle=True
)
val_loader = DataLoader(
    val_dataset,
    batch_size=test_batch_size,
    shuffle=False
)
test_loader = DataLoader(
    test_dataset,
    batch_size=test_batch_size,
    shuffle=False
)
```

We must also declare the value of batch size for training and testing.

DataLoader is an iterator, so we can get a sample (image_path, label) and visualize it.

```python
train_features, train_labels = next(iter(train_loader))
print(f'Feature batch shape: {train_features.size()}')
print(f'Labels batch shape: {train_labels.size()}')
img = train_features[0].permute(1, 2, 0)
label = train_labels[0].item()
plt.imshow(img)
plt.axis('off')
plt.title(f'Label: {classes[label]}')
plt.show()
```

```
Feature batch shape: torch.Size([64, 3, 224, 224])
Labels batch shape: torch.Size([64])
```



DenseNet

❖ Step 6: DenseNet Architecture

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

DenseNet

❖ Step 6: DenseNet Architecture Visualization

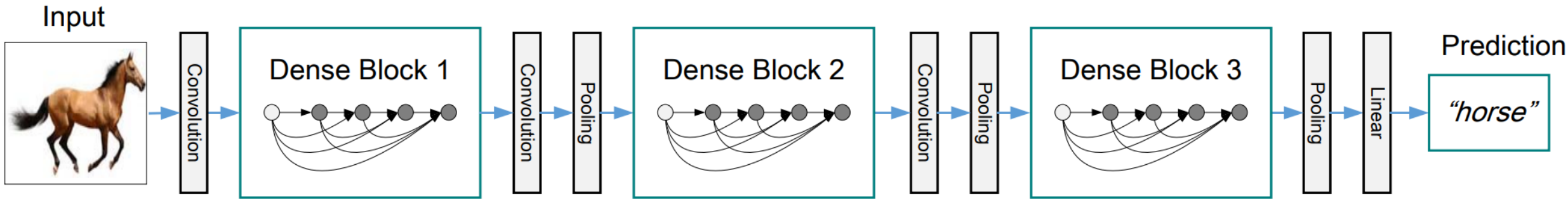
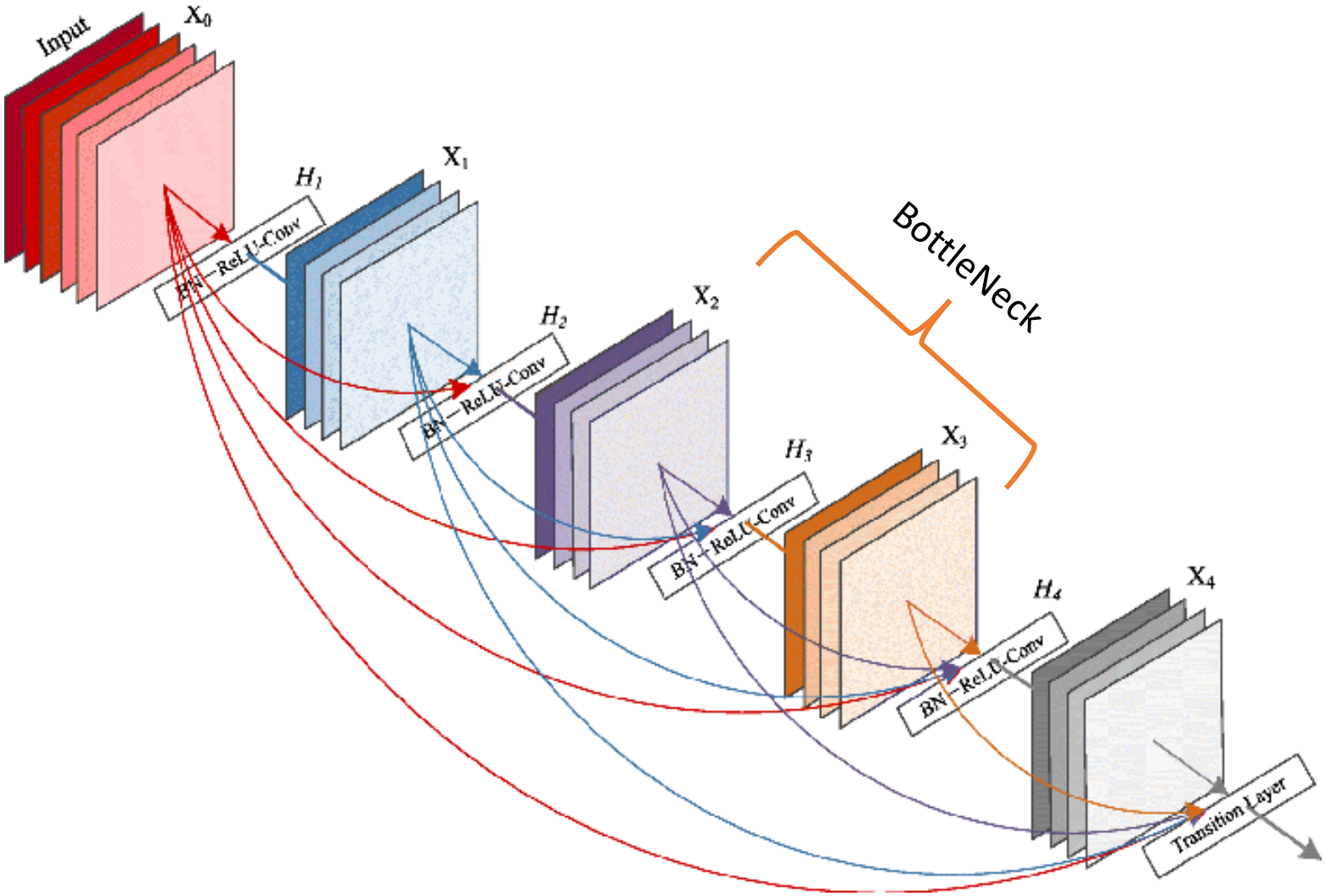


Figure 2: A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling.

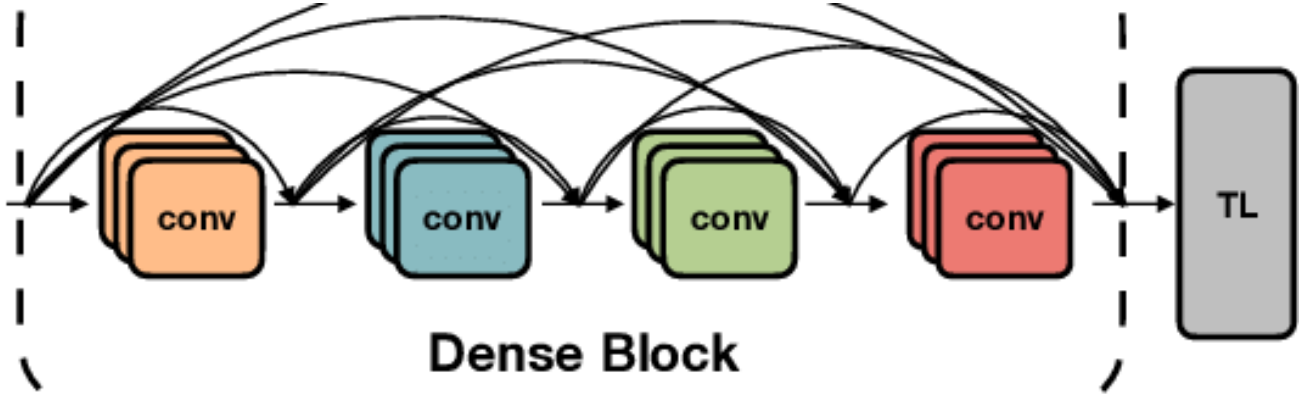
DenseNet

❖ Step 6: DenseBlock Architecture



DenseNet

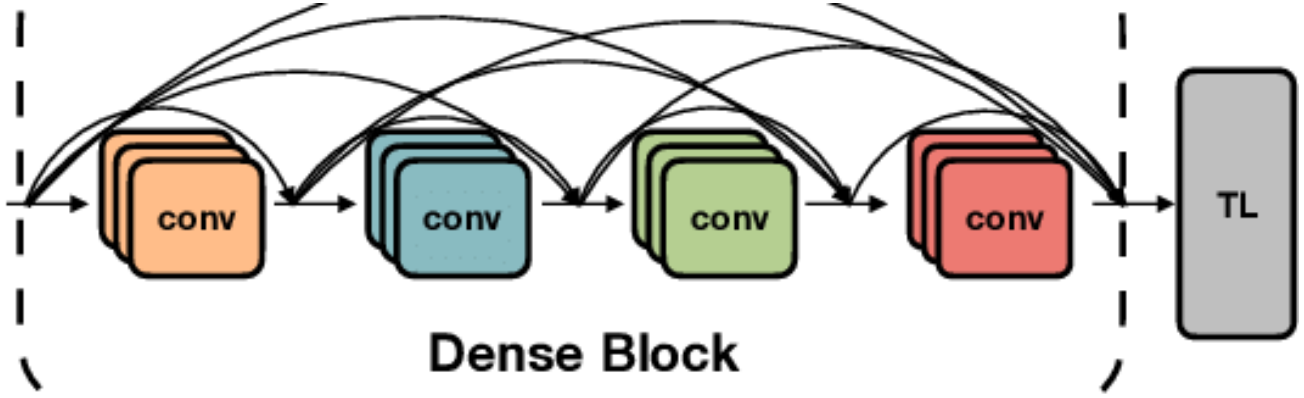
❖ Step 6: Implement DenseBlock




```python
class BottleneckBlock(nn.Module):
    def __init__(self, in_channels, growth_rate):
        super(BottleneckBlock, self).__init__()
        self.bn1 = nn.BatchNorm2d(in_channels)
        self.conv1 = nn.Conv2d(
            in_channels, 4 * growth_rate,
            kernel_size=1,
            bias=False
        )
        self.bn2 = nn.BatchNorm2d(4 * growth_rate)
        self.conv2 = nn.Conv2d(
            4 * growth_rate,
            growth_rate,
            kernel_size=3,
            padding=1,
            bias=False
        )
        self.relu = nn.ReLU()

    def forward(self, x):
        res = x.clone().detach()
        x = self.bn1(x)
        x = self.relu(x)
        x = self.conv1(x)
        x = self.bn2(x)
        x = self.relu(x)
        x = self.conv2(x)
        x = torch.cat([res, x], 1)

        return x
```

DenseNet

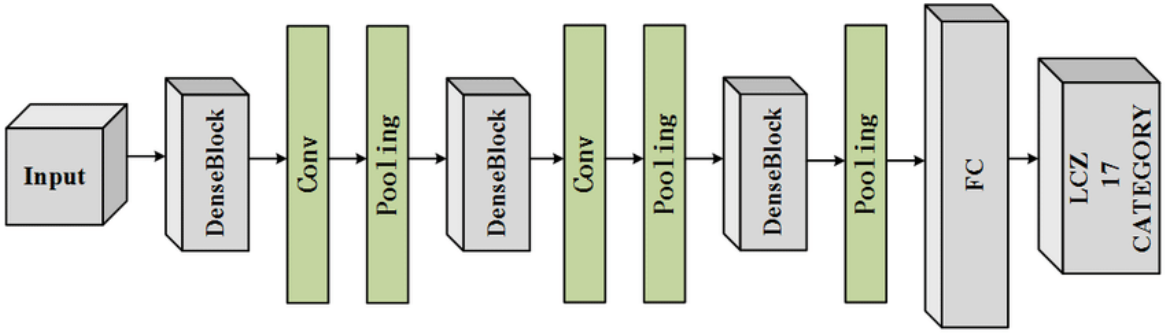
❖ Step 6: Implement DenseBlock

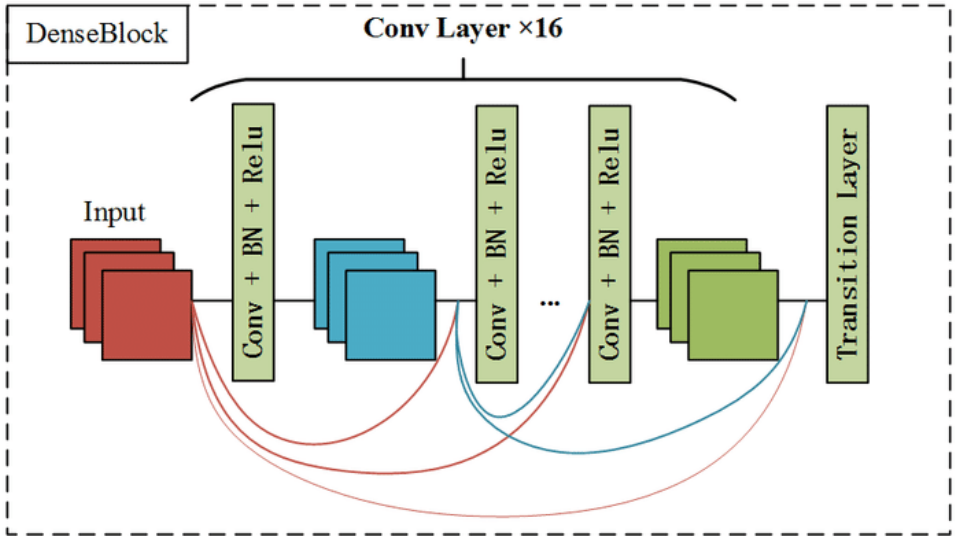



```python
class DenseBlock(nn.Module):
    def __init__(self, num_layers, in_channels, growth_rate):
        super(DenseBlock, self).__init__()
        layers = []
        for i in range(num_layers):
            layers.append(
                BottleneckBlock(
                    in_channels + i * growth_rate,
                    growth_rate
                )
            )
        self.block = nn.Sequential(*layers)

    def forward(self, x):
        return self.block(x)
```

DenseNet

❖ Step 6: Implement DenseNet

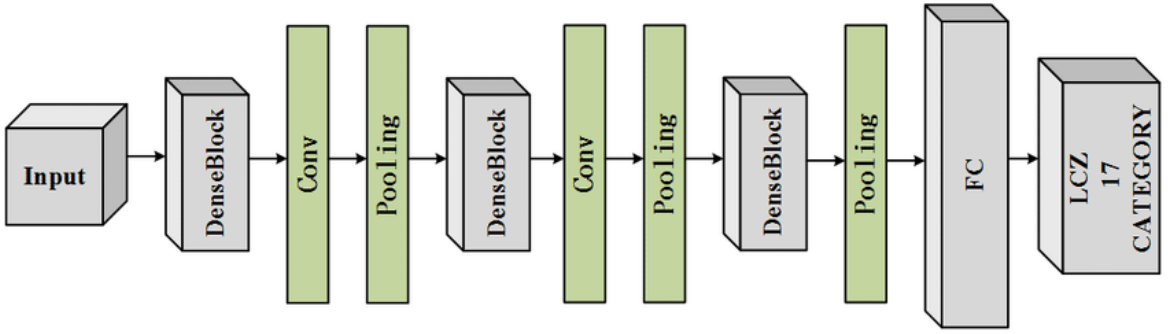


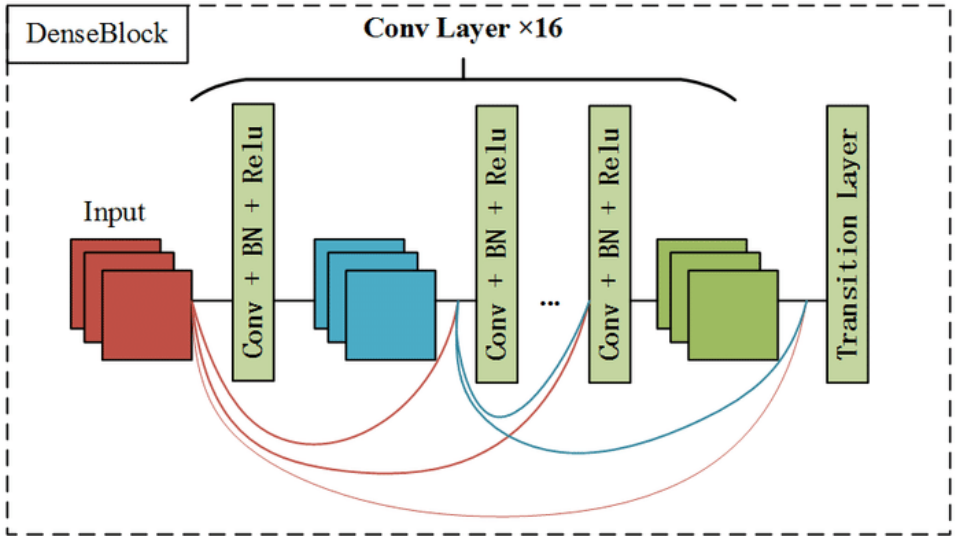



```python
class DenseNet(nn.Module):
    def __init__(self, num_blocks, growth_rate, num_classes):
        super(DenseNet, self).__init__()
        self.conv1 = nn.Conv2d(
            3, 2 * growth_rate,
            kernel_size=7, padding=3,
            stride=2, bias=False
        )
        self.bn1 = nn.BatchNorm2d(2 * growth_rate)
        self.pool1 = nn.MaxPool2d(
            kernel_size=3, stride=2, padding=1
        )

        self.dense_blocks = nn.ModuleList()
        in_channels = 2 * growth_rate
        for i, num_layers in enumerate(num_blocks):
            self.dense_blocks.append(
                DenseBlock(
                    num_layers, in_channels, growth_rate
                )
            )
            in_channels += num_layers * growth_rate
            if i != len(num_blocks) - 1:
                out_channels = in_channels // 2
                self.dense_blocks.append(nn.Sequential(
                    nn.BatchNorm2d(in_channels),
                    nn.Conv2d(
                        in_channels, out_channels,
                        kernel_size=1, bias=False
                    ),
                    nn.AvgPool2d(kernel_size=2, stride=2)
                ))
                in_channels = out_channels
```

DenseNet

❖ Step 6: Implement DenseNet

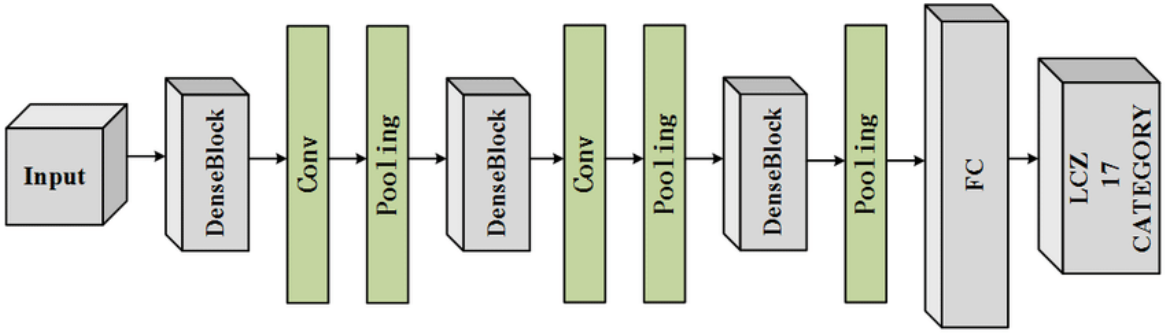


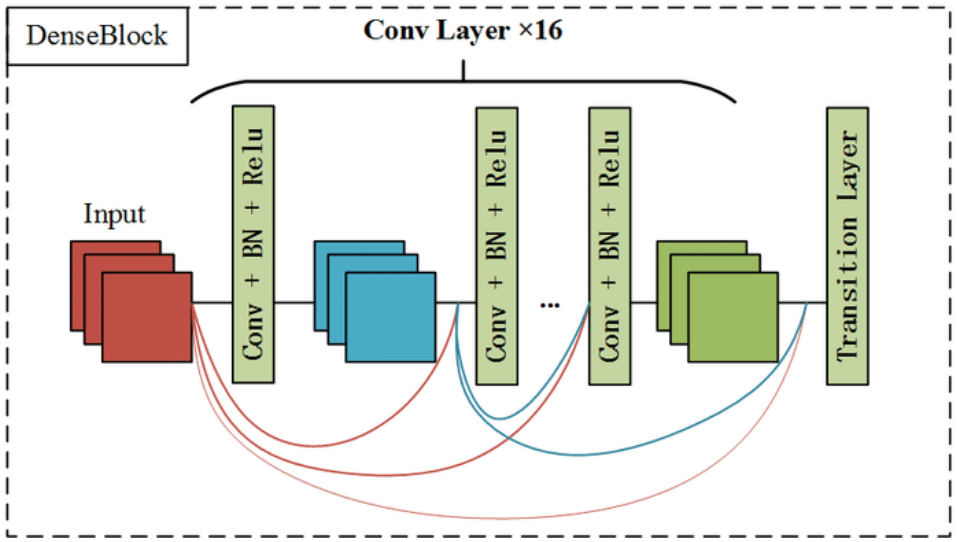



```python
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.pool1(x)

        for block in self.dense_blocks:
            x = block(x)

        x = self.bn2(x)
        x = self.relu(x)
        x = self.pool2(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

DenseNet

❖ Step 6: Implement DenseNet






```python
n_classes = len(list(classes.keys()))
device = 'cuda' if torch.cuda.is_available() else 'cpu'

model = DenseNet(
    [6, 12, 24, 16],
    growth_rate=32,
    num_classes=n_classes
).to(device)
```

```python
model.eval()

dummy_tensor = torch.randn(1, 3, 224, 224).to(device)

with torch.no_grad():
    output = model(dummy_tensor)

print('Output shape:', output.shape)
```

```
Output shape: torch.Size([1, 6])
```

DenseNet

❖ Step 8, 9: Create fit and evaluate function

```python
def evaluate(model, dataloader, criterion, device):
    model.eval()
    correct = 0
    total = 0
    losses = []
    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            losses.append(loss.item())
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    loss = sum(losses) / len(losses)
    acc = correct / total

    return loss, acc
```

```python
def fit(
    model,
    train_loader,
    val_loader,
    criterion,
    optimizer,
    device,
    epochs
):
    train_losses = []
    val_losses = []

    for epoch in range(epochs):
        batch_train_losses = []

        model.train()
        for idx, (inputs, labels) in enumerate(train_loader):
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            batch_train_losses.append(loss.item())

        train_loss = sum(batch_train_losses) / len(batch_train_losses)
        train_losses.append(train_loss)

        val_loss, val_acc = evaluate(
            model, val_loader,
            criterion, device
        )
        val_losses.append(val_loss)

        print(f'EPOCH {epoch + 1}:\tTrain loss: {train_loss:.4f}\tVal loss: {val_loss:.4f}')

    return train_losses, val_losses
```

DenseNet

❖ Step 10: Training

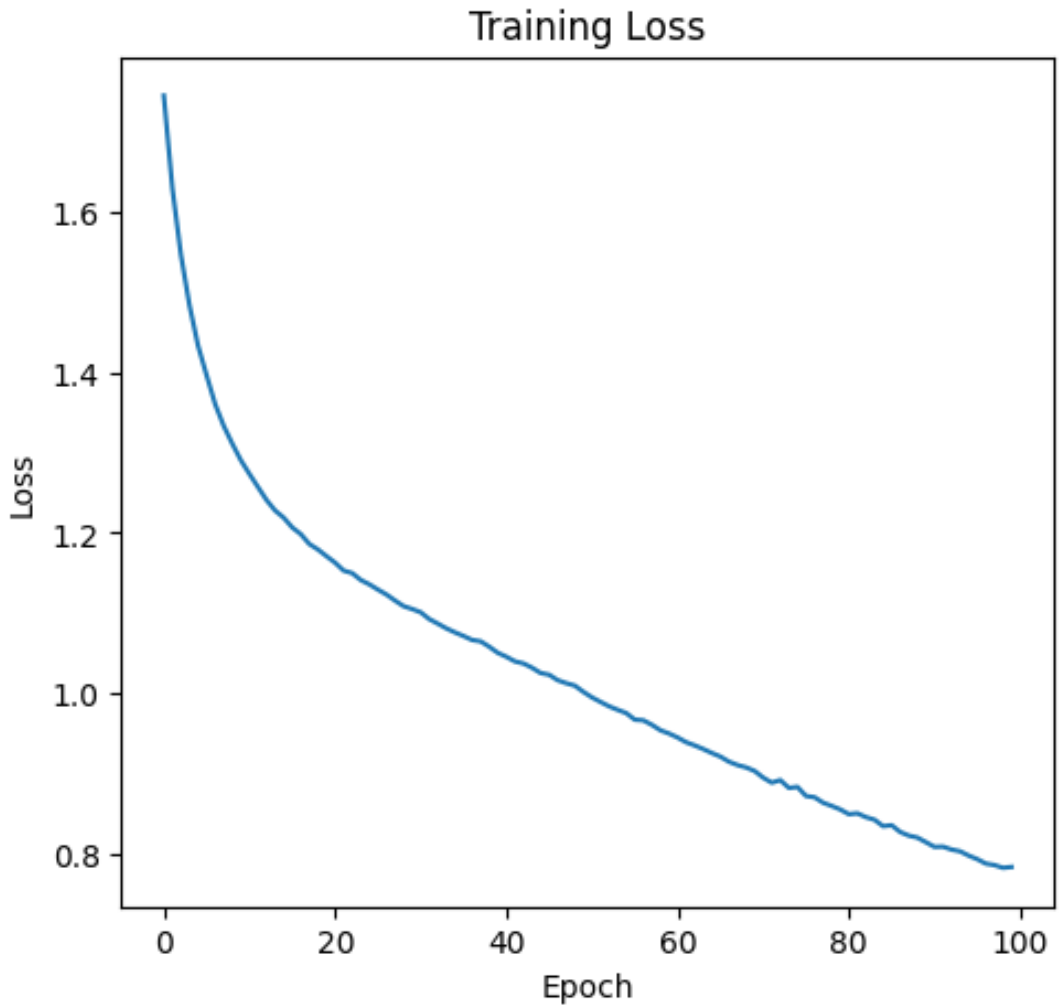
```
lr = 1e-3
epochs = 100

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(
    model.parameters(),
    lr=lr
)
```

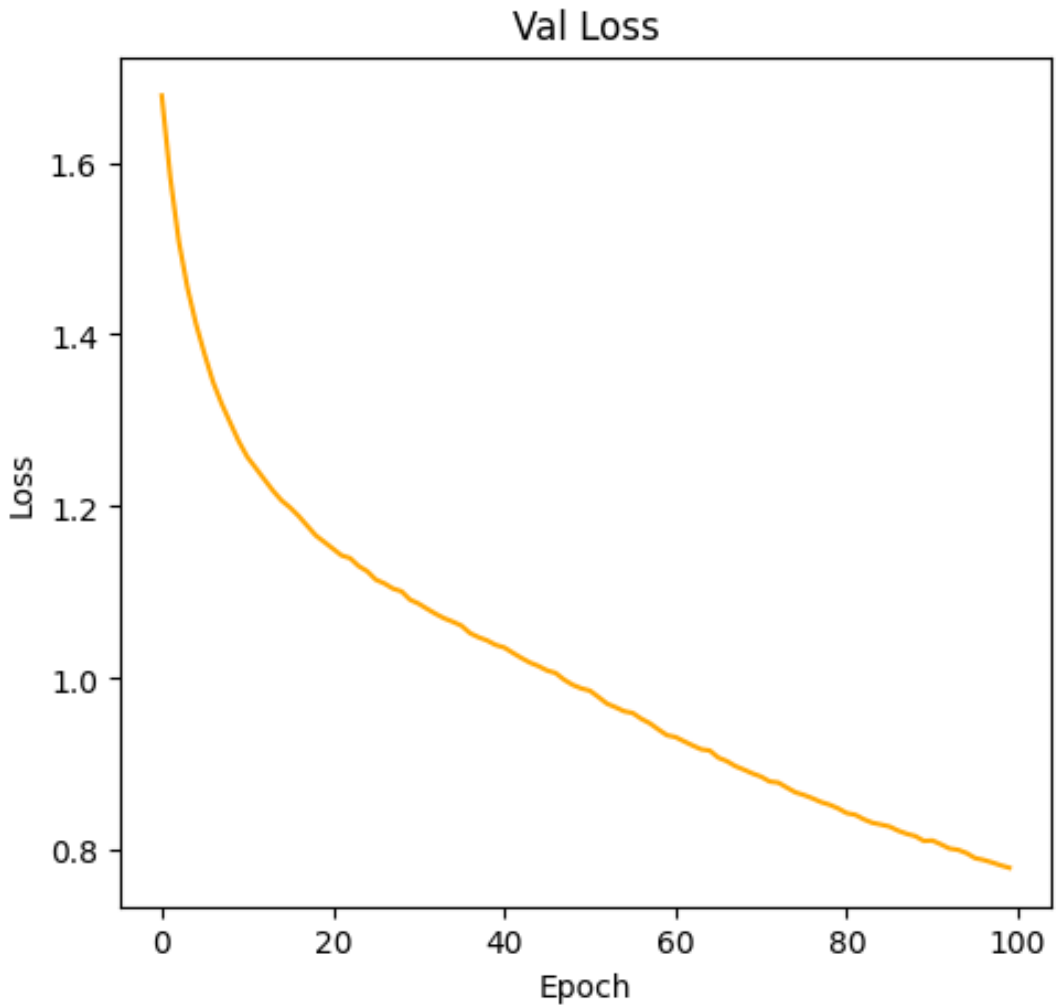
```
train_losses, val_losses = fit(
    model,
    train_loader,
    val_loader,
    criterion,
    optimizer,
    device,
    epochs
)
```

```
EPOCH 1:        Train loss: 1.7440      Val loss: 1.6777
EPOCH 2:        Train loss: 1.6288      Val loss: 1.5821
EPOCH 3:        Train loss: 1.5459      Val loss: 1.5079
EPOCH 4:        Train loss: 1.4831      Val loss: 1.4544
EPOCH 5:        Train loss: 1.4330      Val loss: 1.4122
EPOCH 6:        Train loss: 1.3960      Val loss: 1.3770
EPOCH 7:        Train loss: 1.3602      Val loss: 1.3445
EPOCH 8:        Train loss: 1.3331      Val loss: 1.3194
EPOCH 9:        Train loss: 1.3110      Val loss: 1.2973
EPOCH 10:       Train loss: 1.2904      Val loss: 1.2756
EPOCH 11:       Train loss: 1.2734      Val loss: 1.2575
EPOCH 12:       Train loss: 1.2571      Val loss: 1.2445
EPOCH 13:       Train loss: 1.2409      Val loss: 1.2312
EPOCH 14:       Train loss: 1.2279      Val loss: 1.2182
EPOCH 15:       Train loss: 1.2190      Val loss: 1.2069
EPOCH 16:       Train loss: 1.2069      Val loss: 1.1987
EPOCH 17:       Train loss: 1.1985      Val loss: 1.1888
EPOCH 18:       Train loss: 1.1863      Val loss: 1.1775
EPOCH 19:       Train loss: 1.1795      Val loss: 1.1662
```

DenseNet

❖ Step 10: Training





DenseNet

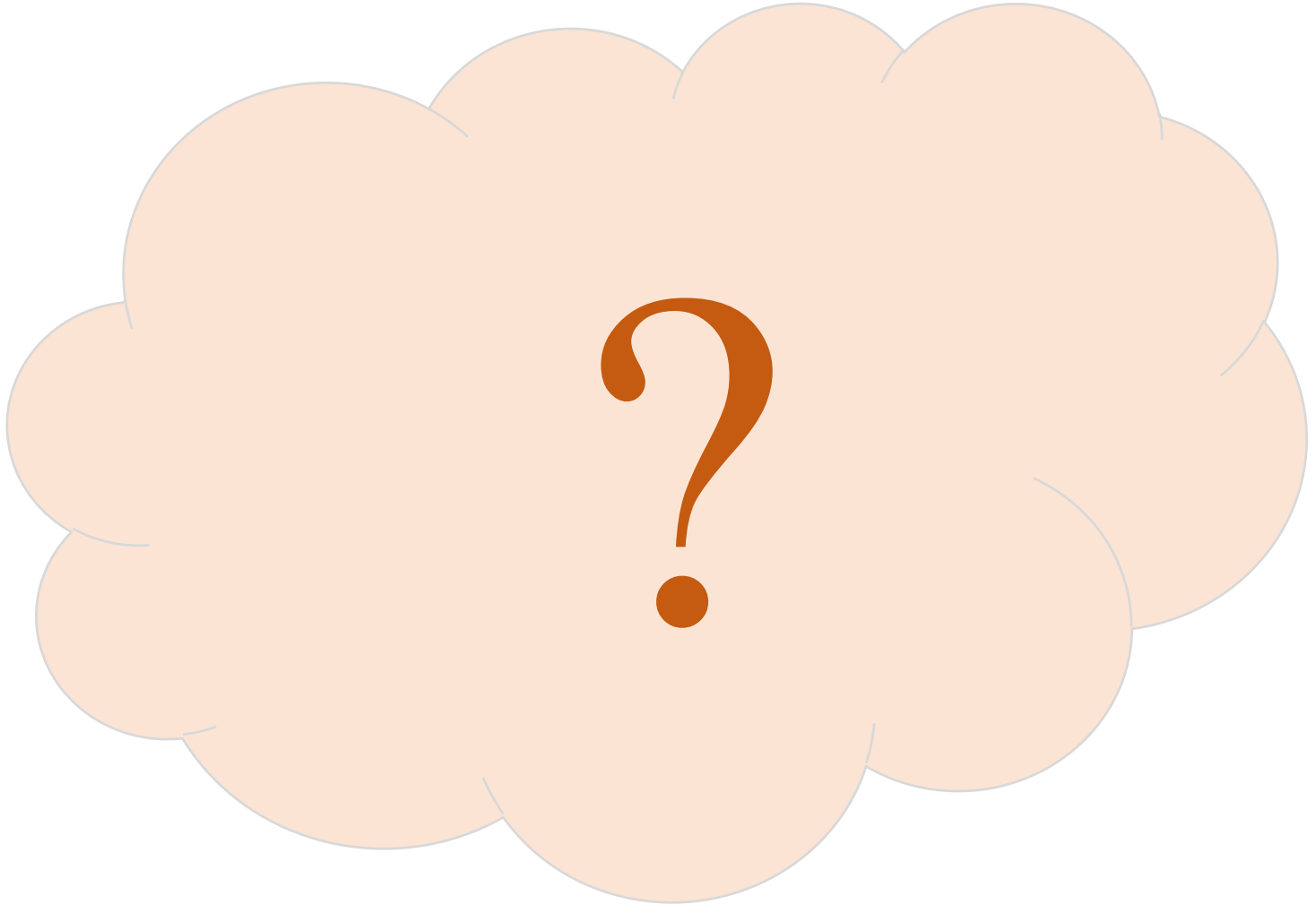
❖ Step 10: Training

```python
def evaluate(model, dataloader, criterion, device):
    model.eval()
    correct = 0
    total = 0
    losses = []
    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            losses.append(loss.item())
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    loss = sum(losses) / len(losses)
    acc = correct / total

    return loss, acc
```

```python
val_loss, val_acc = evaluate(
    model,
    val_loader,
    criterion,
    device
)
test_loss, test_acc = evaluate(
    model,
    test_loader,
    criterion,
    device
)

print('Evaluation on val/test dataset')
print('Val accuracy: ', val_acc)
print('Test accuracy: ', test_acc)
```

```
Evaluation on val/test dataset
Val accuracy:  0.7267545422158889
Test accuracy:  0.7133333333333334
```

Question




Thank you!