Computer Vision

Lecture 04: Convolutional Neural Network-1

Differentiable layers

I ⇨ layer h ⇨ X ⇨ Any differentiable intermediate layer f whose parameter is θ ⇨ y ⇨ layer g ⇨ $Loss\ L$

We need to implement three things for an intermediate layer f :

forward rule:	$y = f(X;\theta)$	for	$g\big(f(X;\theta)\big) = \mathrm{L}$
backward rule:	$\frac{dy}{dX}$	for	$\frac{dL}{dX} = \frac{dy}{dX} \times \frac{dL}{dy}$
parameter update rule:	$\frac{dy}{d\theta}$	for	$\theta^{new} = \theta - \varepsilon \frac{dy}{d\theta} \times \frac{dL}{dy}$

Differentiable layers

I ⇨ layer h ⇨ X ⇨ Any differentiable intermediate layer f whose parameter is θ ⇨ y ⇨ layer g ⇨ $Loss\ \mathrm{L}$

We need to implement three things for an intermediate layer f :

forward rule: $y = f(X;\theta)$ for $g\big(f(X;\theta)\big) = \mathrm{L}$

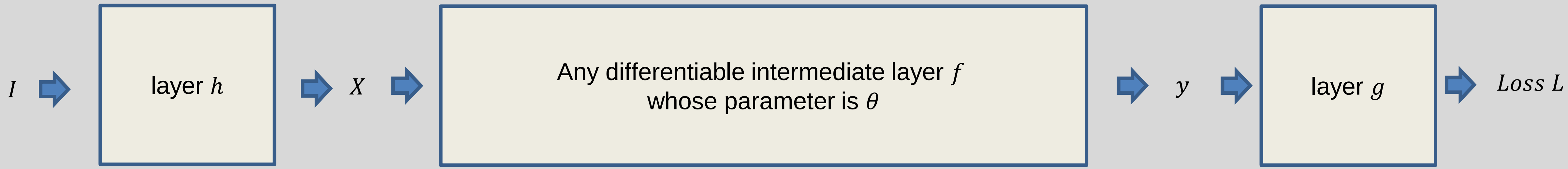
backward rule: $\frac{dy}{dX}$ for $\frac{dL}{dX} = \frac{dy}{dX} \times \frac{dL}{dy}$

parameter update rule: $\frac{dy}{d\theta}$ for $\theta^{new} = \theta - \varepsilon \frac{dy}{d\theta} \times \frac{dL}{dy}$

⇨ PyTorch can do these automatically.

loss.backward()
optimizer.step()

Differentiable layers



Some layers (e.g. pooling, activation) do not have parameters θ. It requires only two:

forward rule: $y = f(X;\theta)$ for $g\big(f(X;\theta)\big) = \mathrm{L}$

backward rule: $\frac{dy}{dX}$ for $\frac{dL}{d\mathrm{X}} = \frac{dy}{d\mathrm{X}} \times \frac{dL}{dy}$

In convolutional layers, parameters are convolutional filter kernel’s weights.

CNNs

RGB image

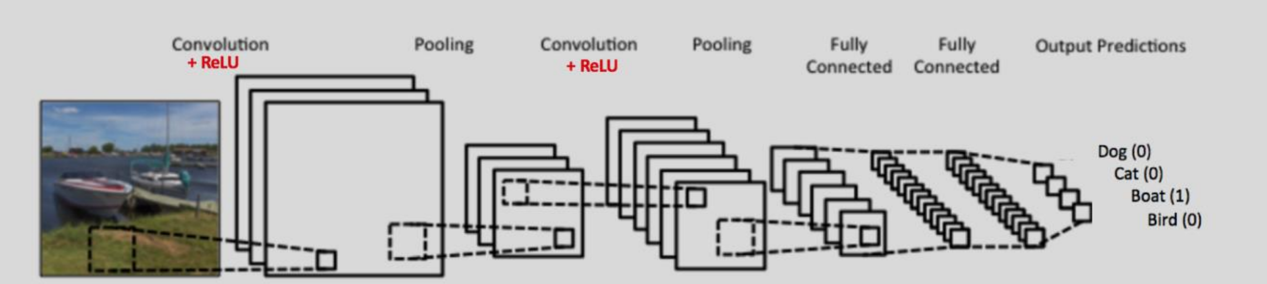
Any Differentiable Layers

cat

Semantic labels

CNNs

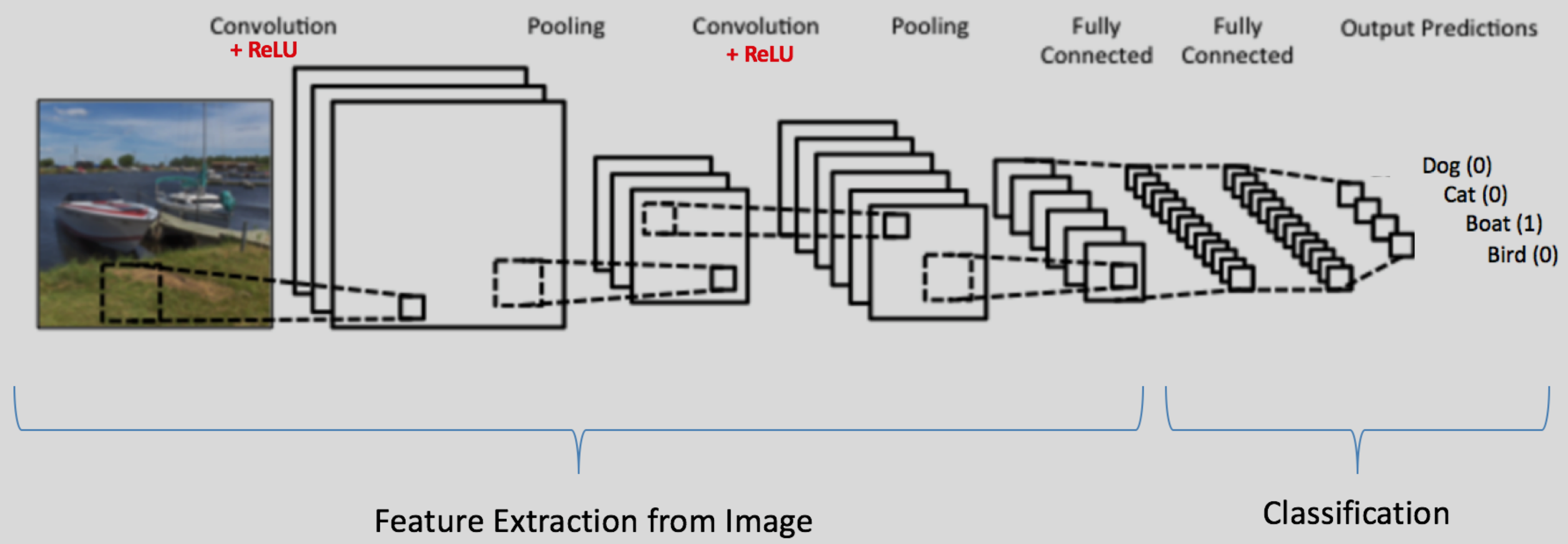
RGB image



cat

Semantic labels

Convolutional neural network



2D Convolution

1	1	0	1	2
2	0	0	1	0
0	0	2	2	1
0	0	0	2	0
0	0	2	1	1

Image (5x5)

0	0	-1
1	-1	0
0	1	-1

Filter kernel (3x3)

2D Convolution

1*0+1*0+0*-1
+ 2*1+0*-1+0*0
+ 0*0+0*1+2*-1

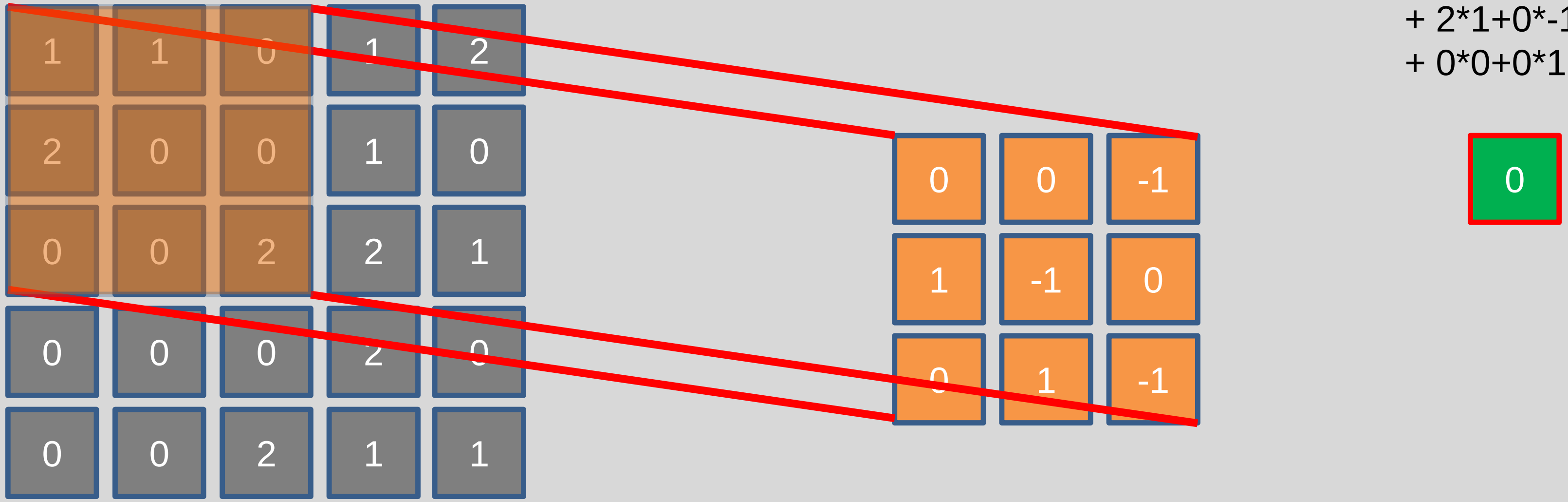
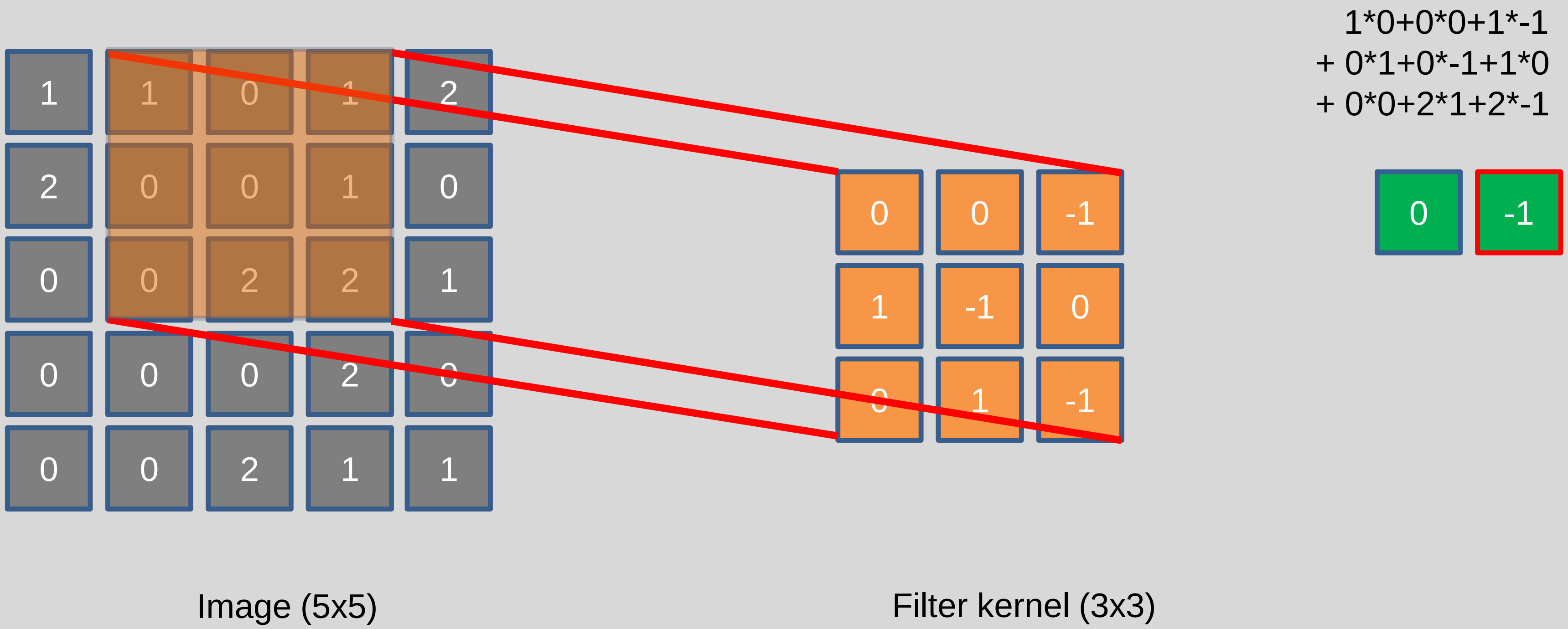


Image (5x5)

Filter kernel (3x3)

2D Convolution



2D Convolution

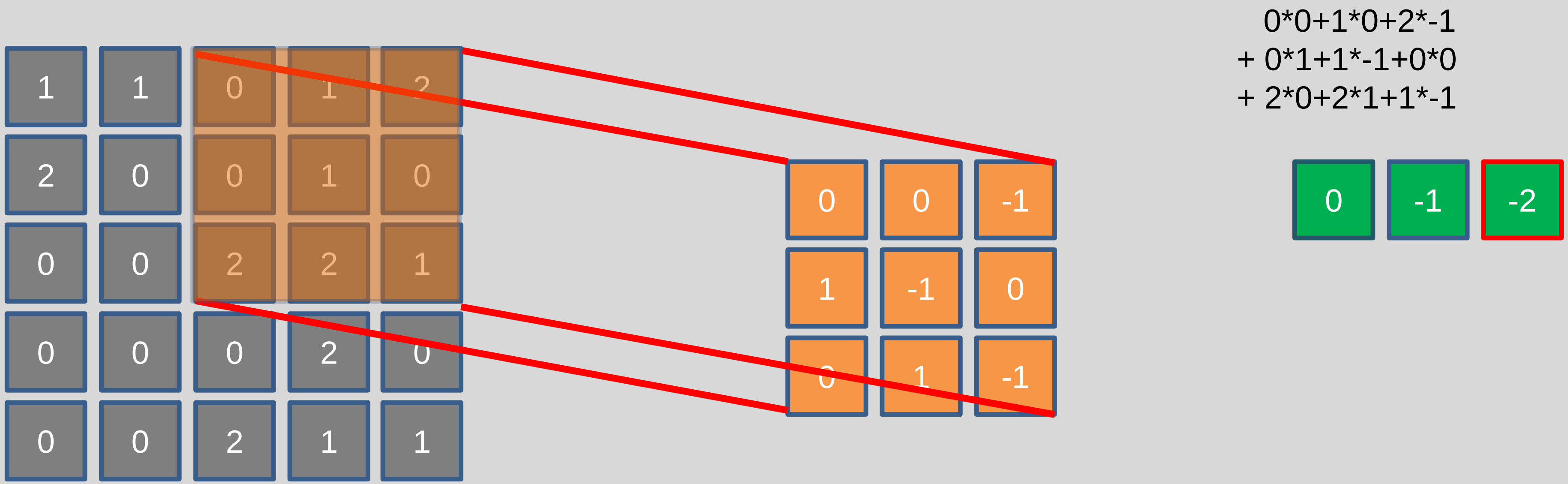


Image (5x5)

Filter kernel (3x3)

2D Convolution

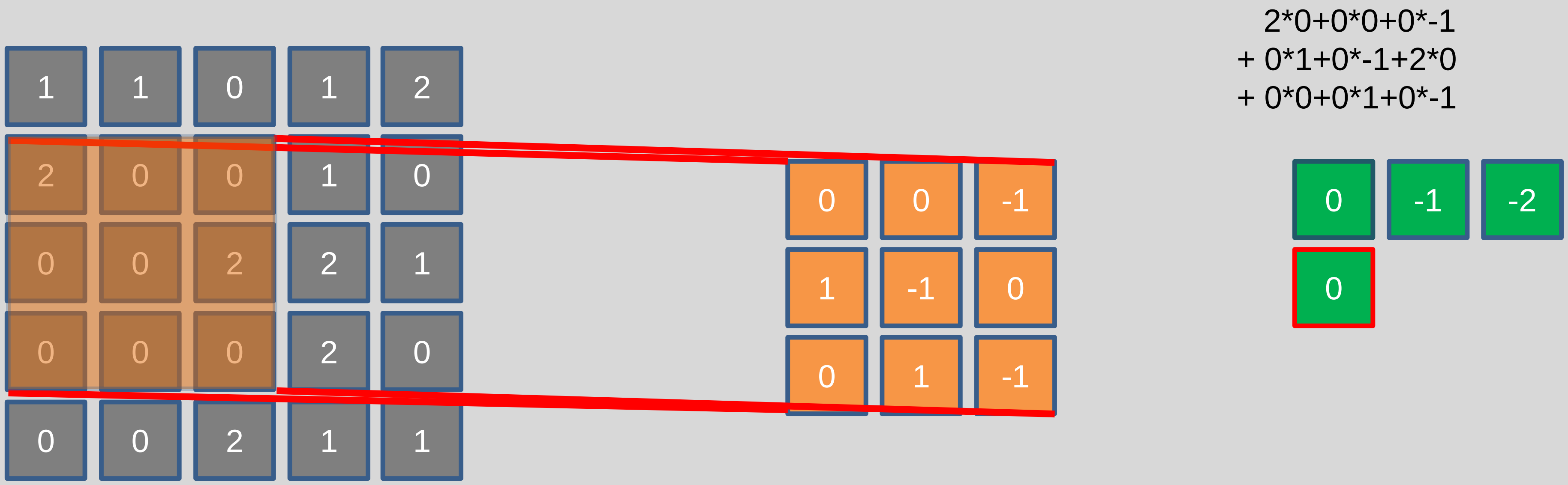


Image (5x5)

Filter kernel (3x3)

2D Convolution

2*0+2*0+1*-1
+ 0*1+2*-1+2*0
+ 2*0+1*1+1*-1

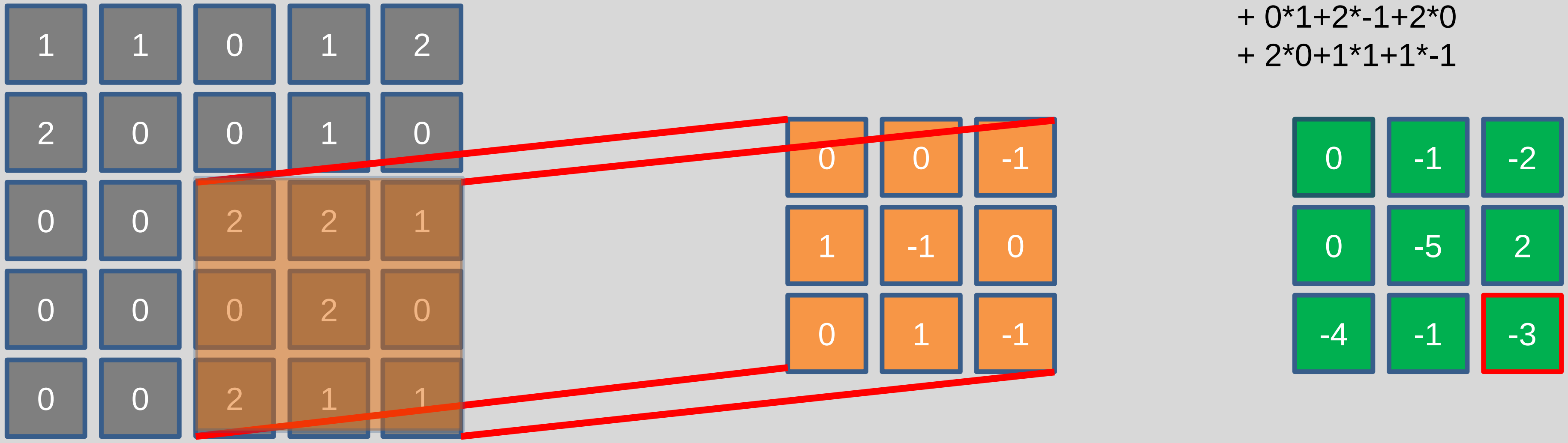


Image (5x5)

Filter kernel (3x3)

Output feature (3x3)

Ex. Image blurring

1	1	1
1	1	1
1	1	1

2D Convolution w/ zero padding

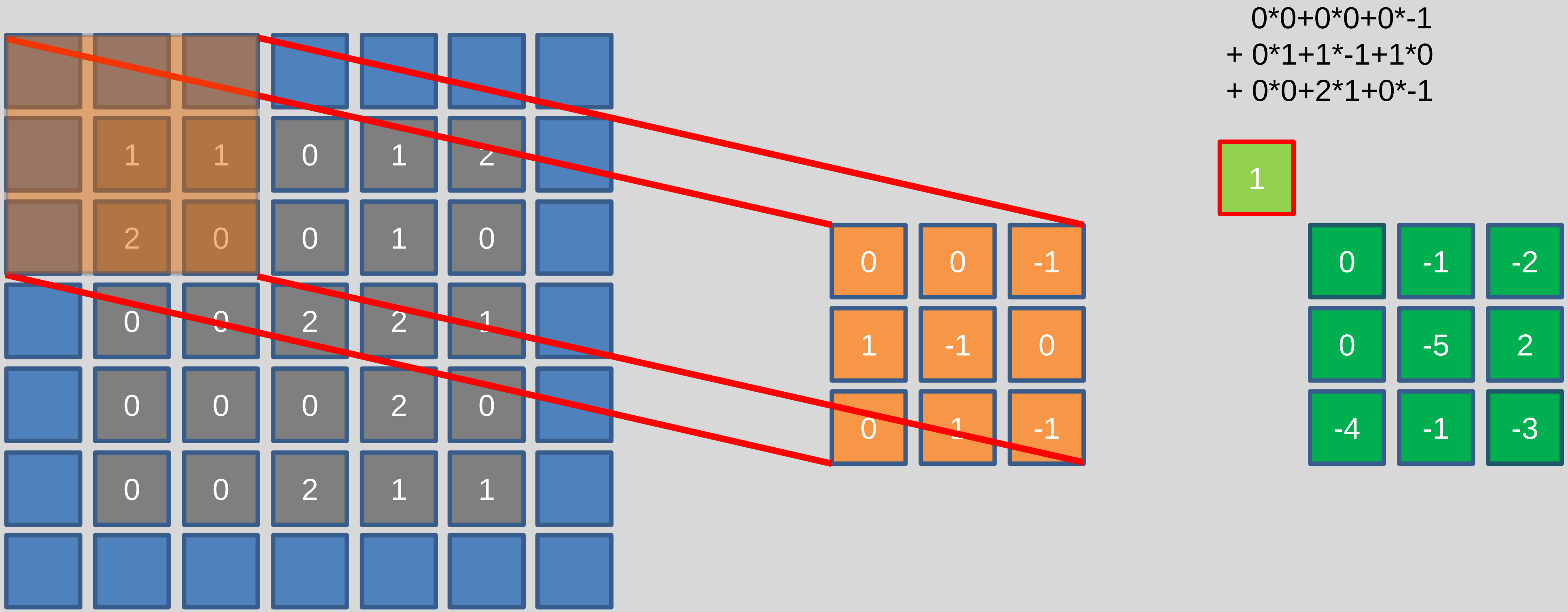


Image (5x5)

Filter kernel (3x3)

2D Convolution w/ zero padding

0*0+0*0+0*-1
+ 1*1+1*-1+0*0
+ 2*0+0*1+0*-1

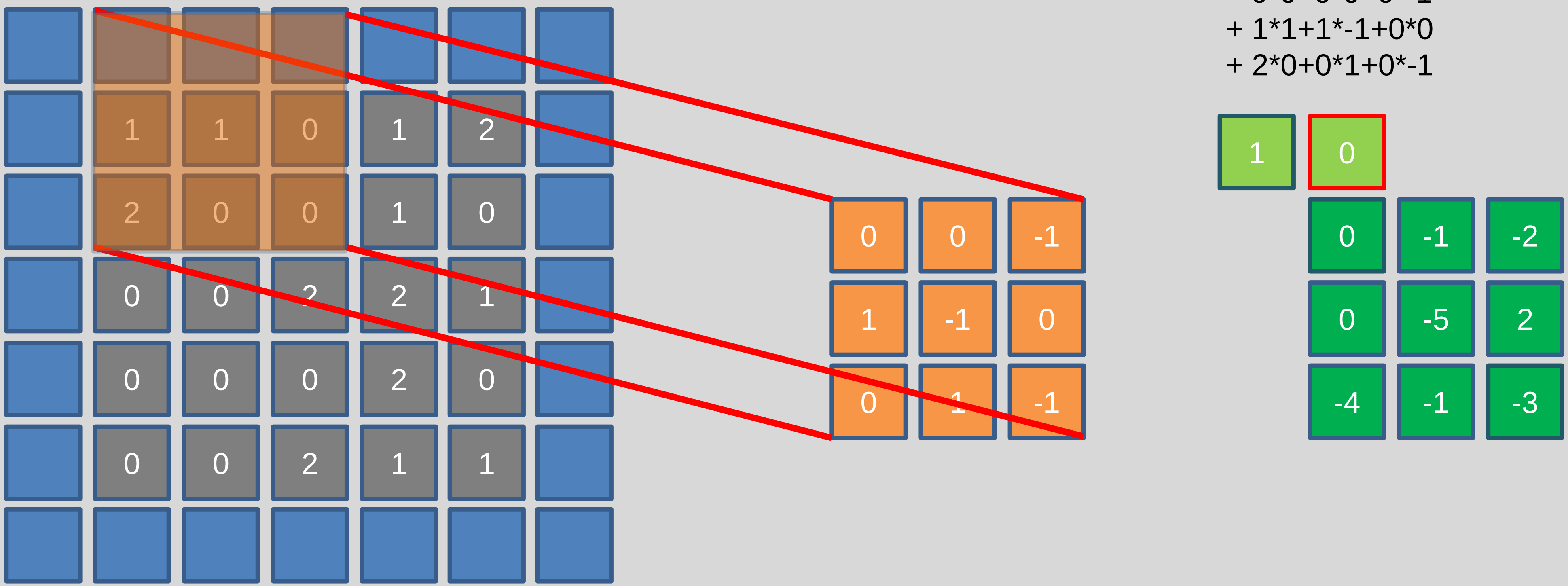


Image (5x5)

Filter kernel (3x3)

2D Convolution w/ zero padding

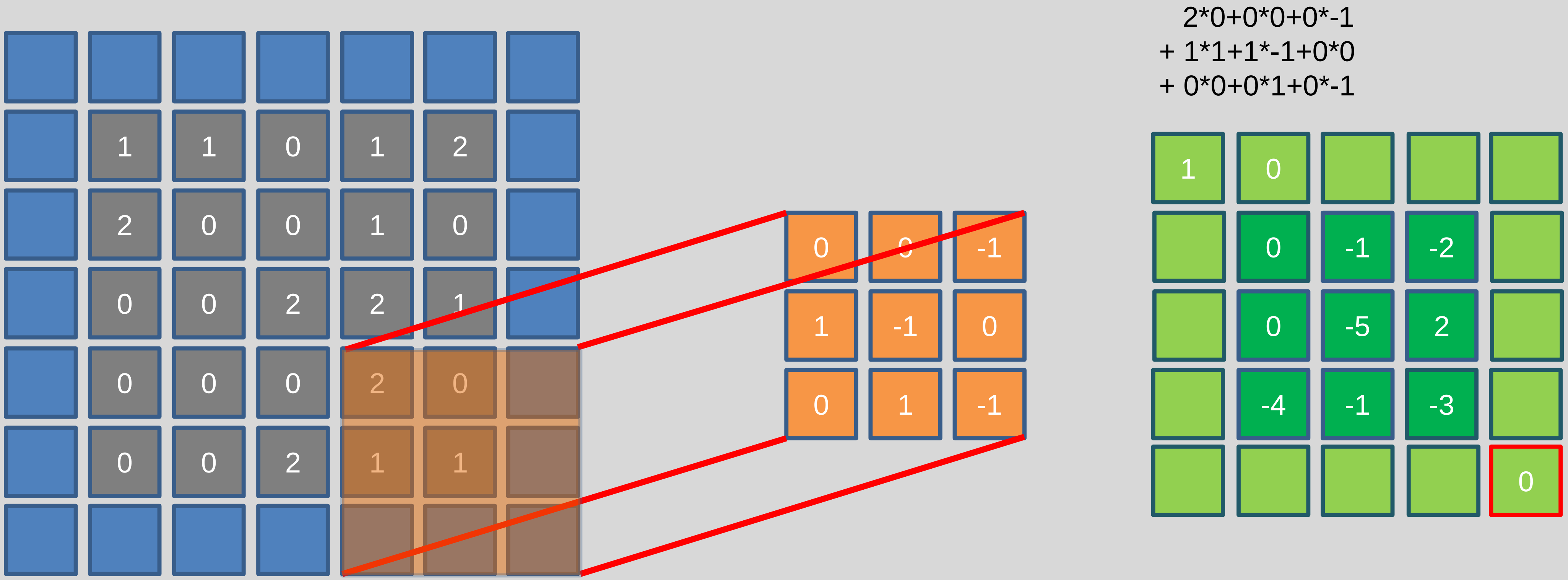


Image (5x5)

Filter kernel (3x3)

Output feature (5x5)

2D Convolution w/ zero padding

→ (K-1) / 2 padding is required to obtain the original size.

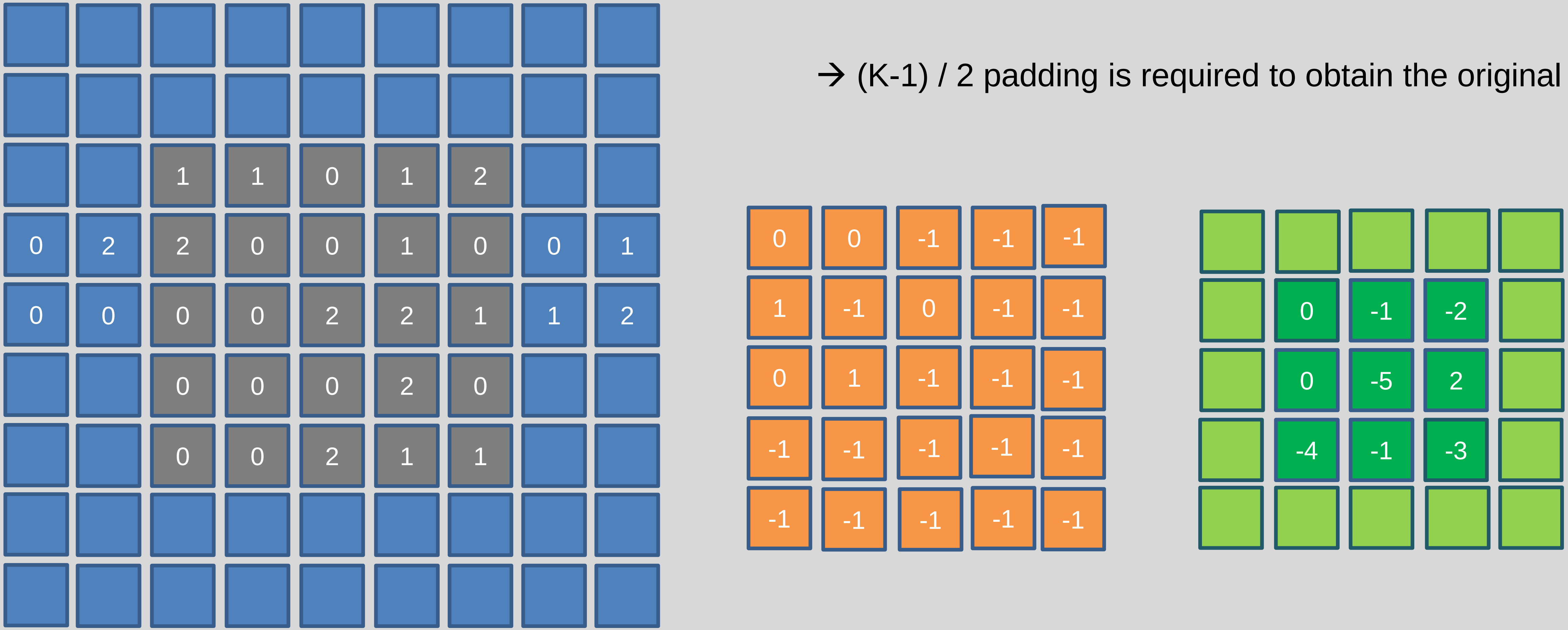
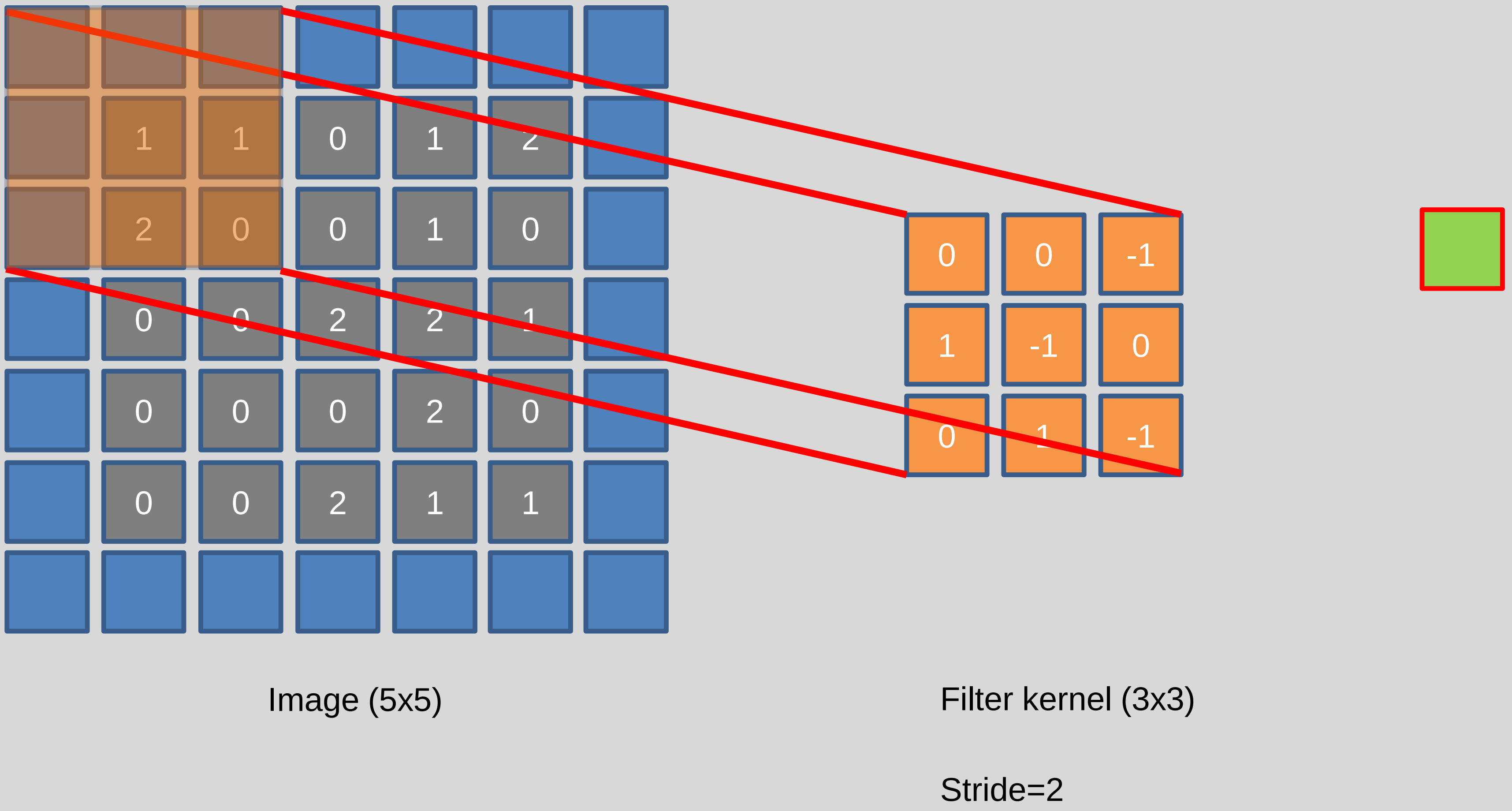


Image (5x5)

Filter kernel (5x5)

Output feature (5x5)

2D Convolution w/ stride



2D Convolution w/ stride

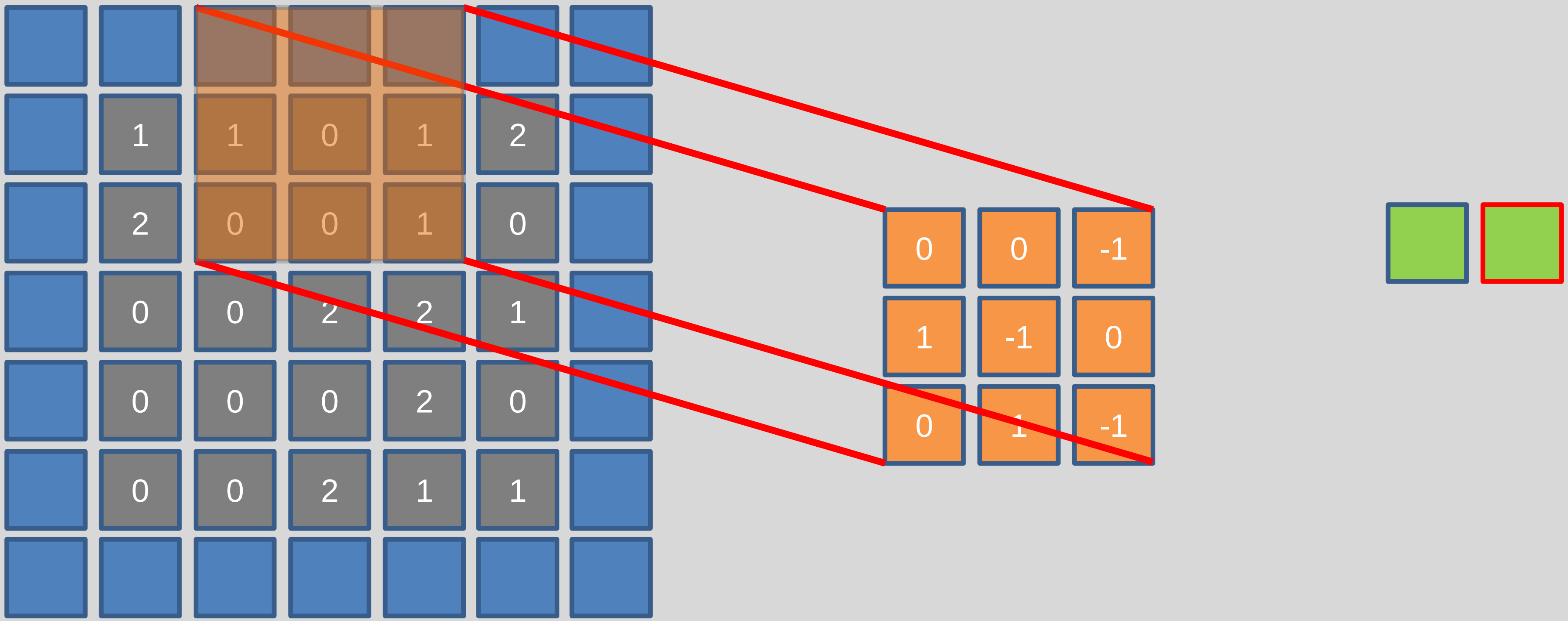


Image (5x5)

Filter kernel (3x3)

Stride=2

2D Convolution w/ stride

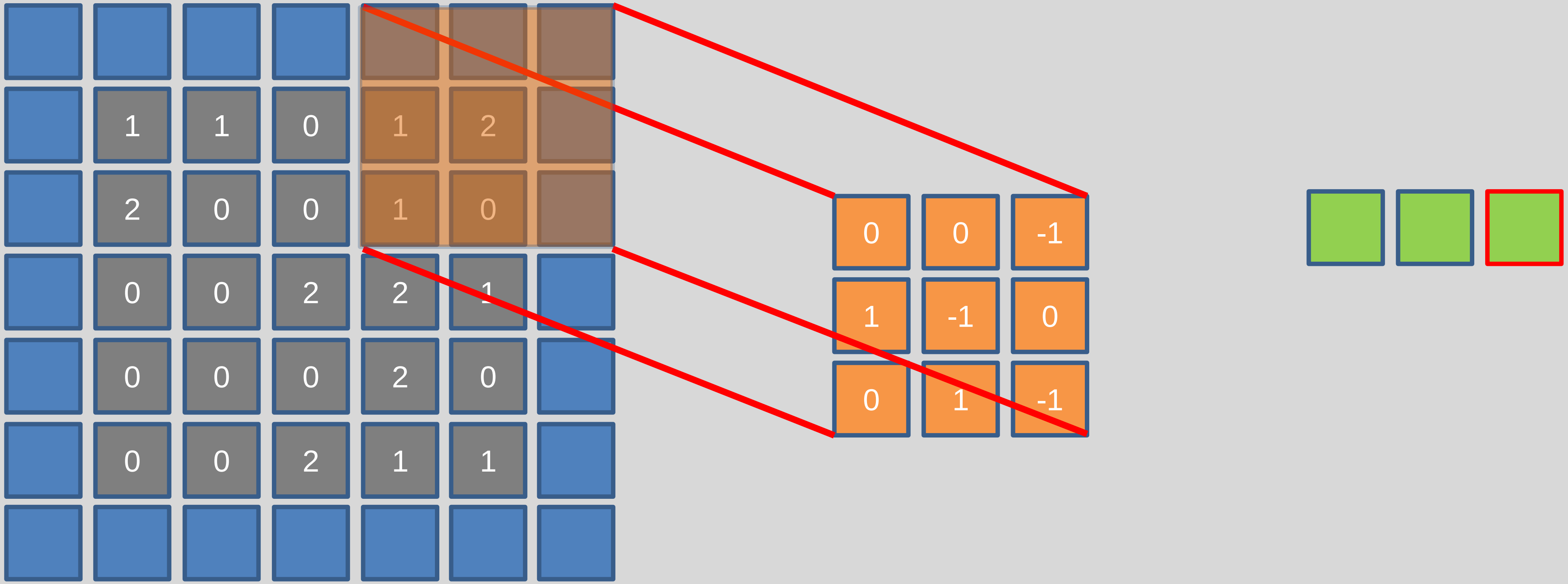


Image (5x5)

Filter kernel (3x3)

Stride=2

2D Convolution w/ stride

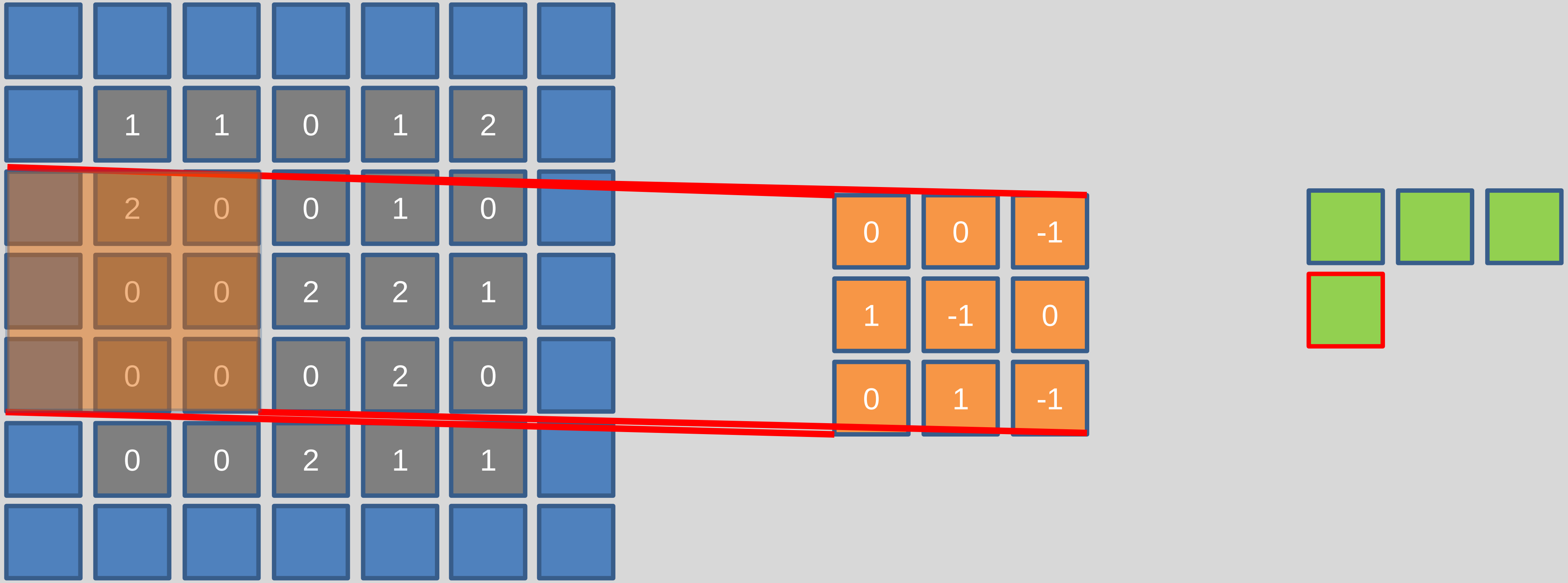


Image (5x5)

Filter kernel (3x3)

Stride=2

2D Convolution w/ stride

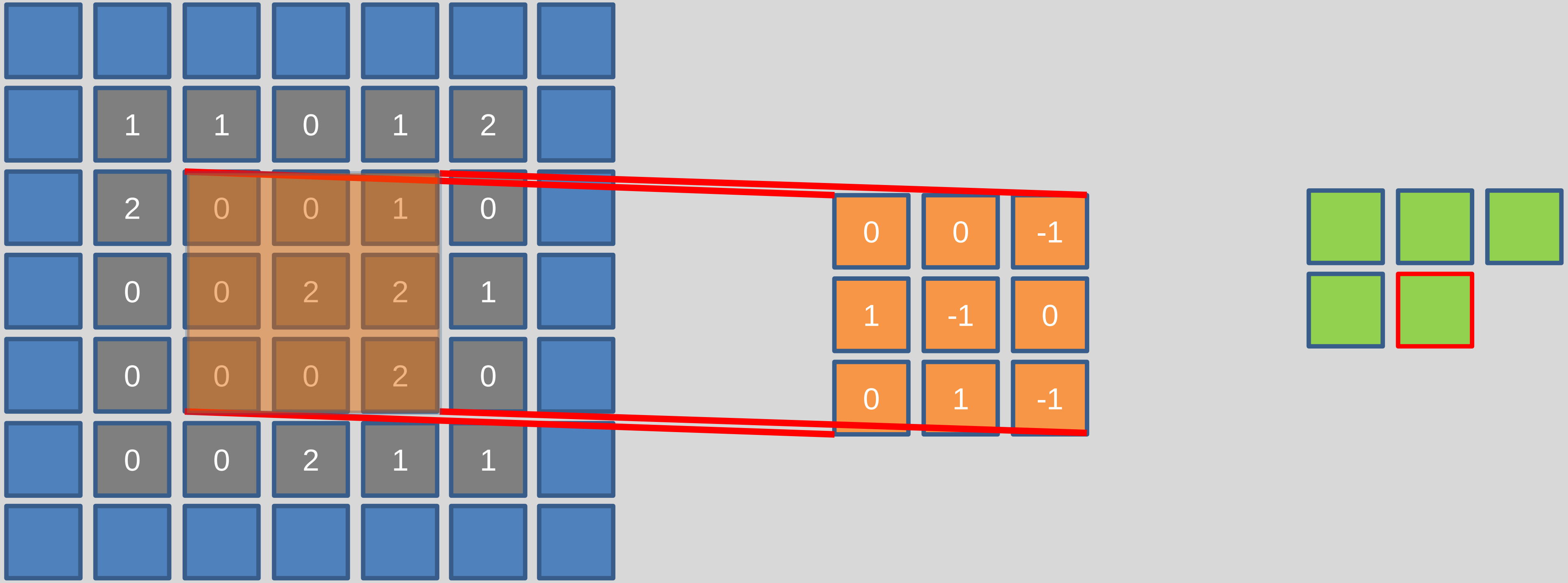


Image (5x5)

Filter kernel (3x3)

Stride=2

2D Convolution w/ stride

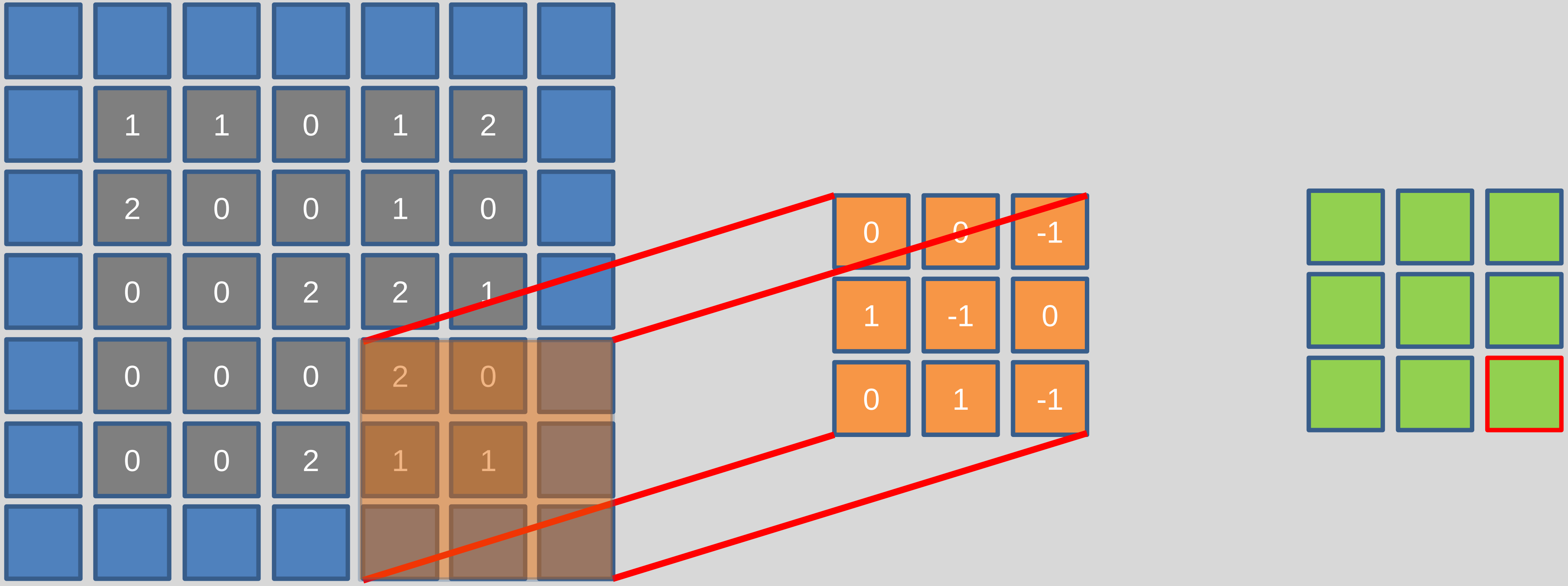
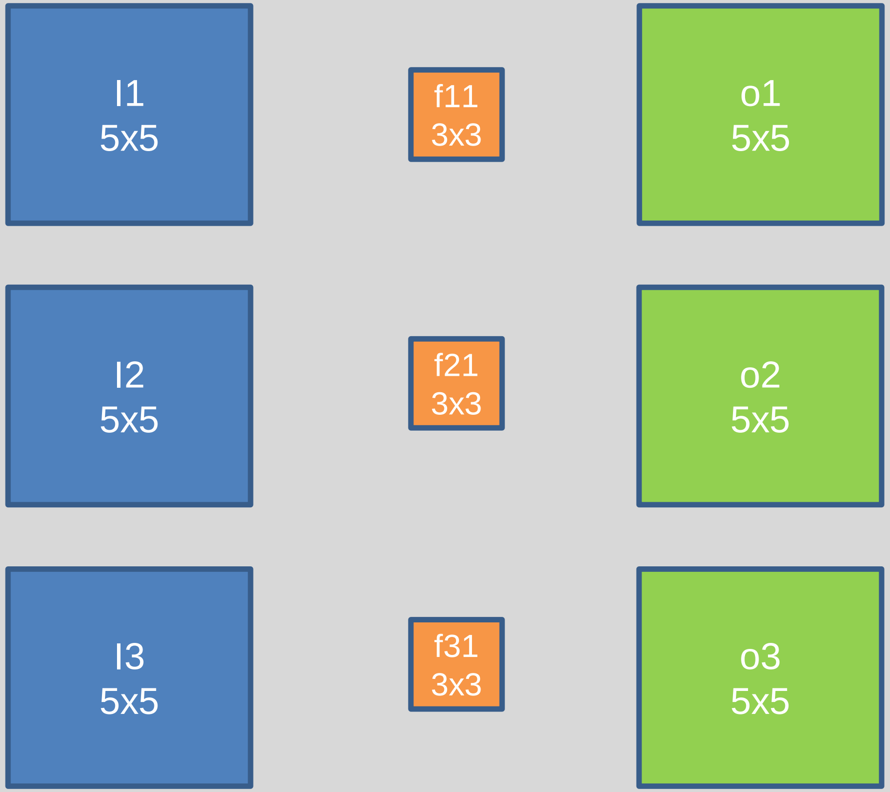


Image (5x5)

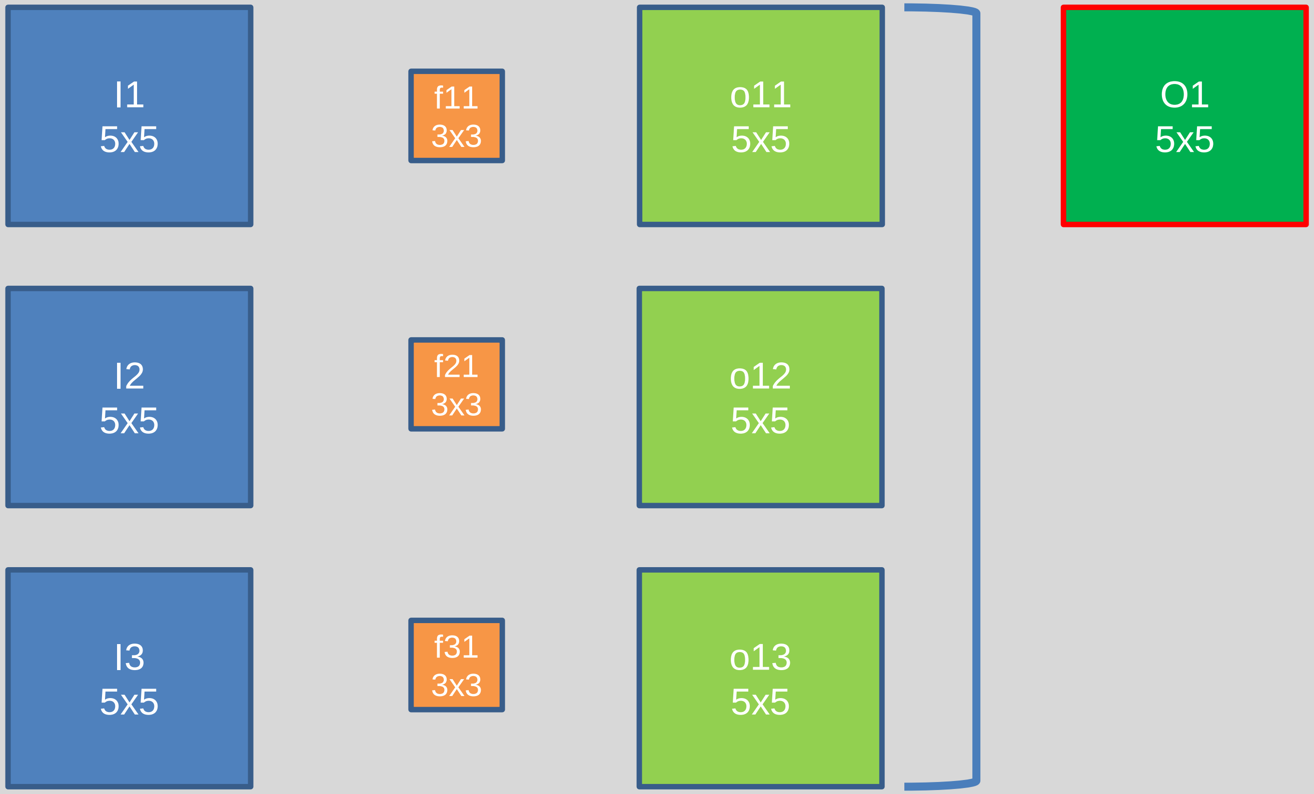
Filter kernel (3x3)

Stride=2

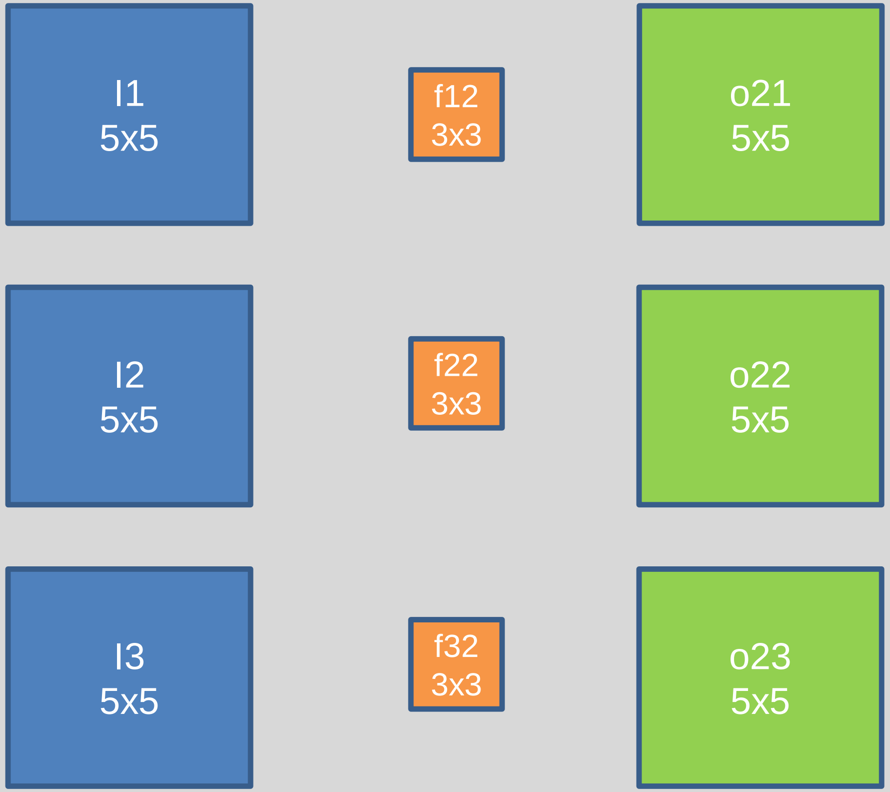
Convolutional layer



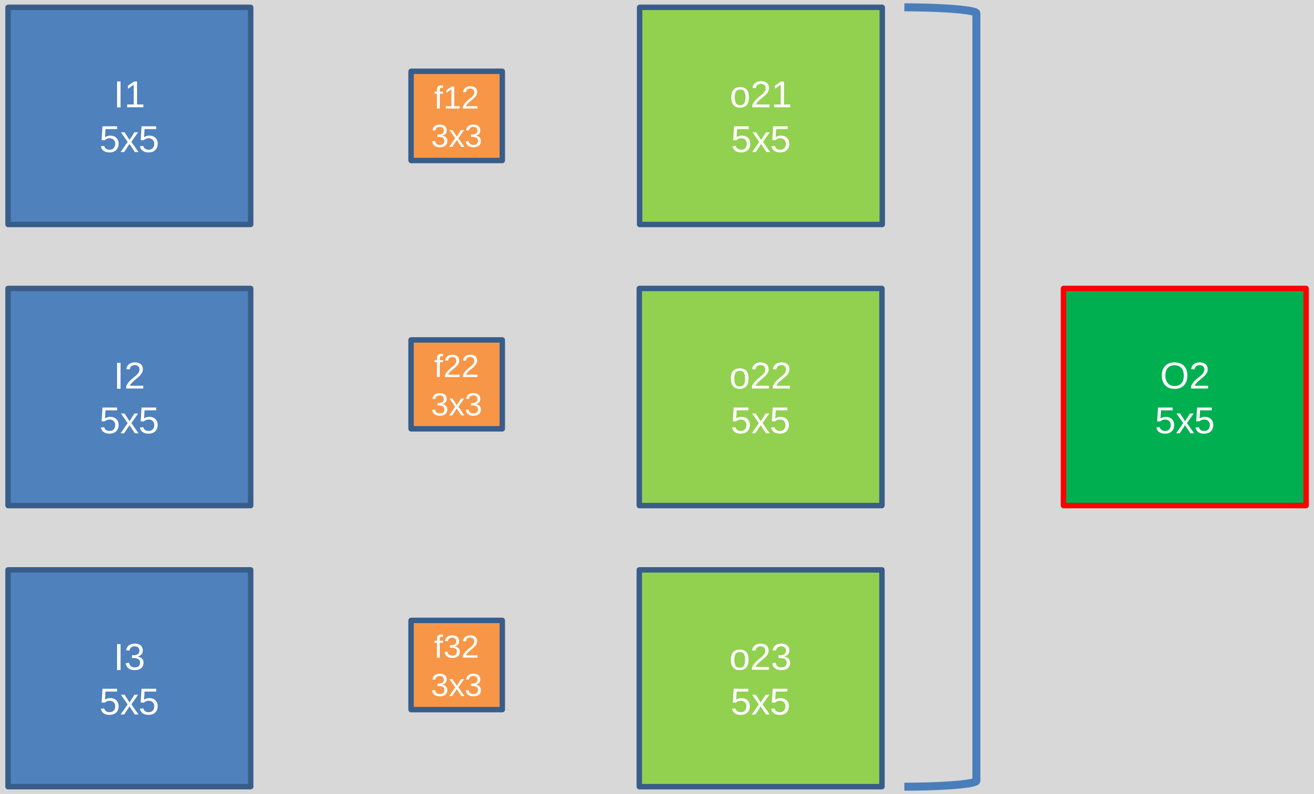
Convolutional layer



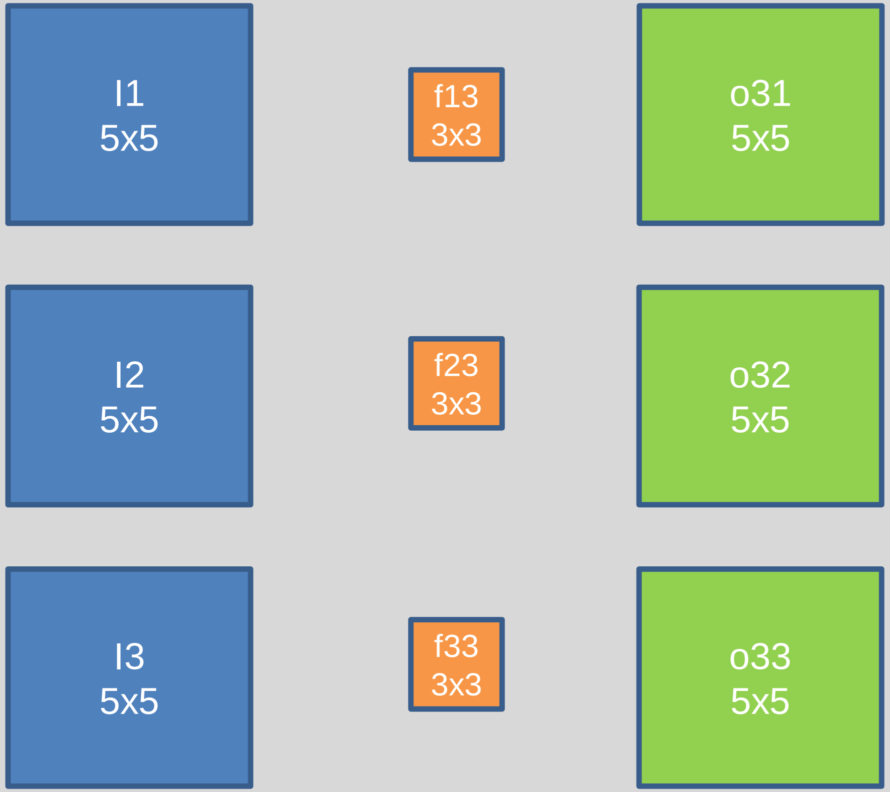
Convolutional layer



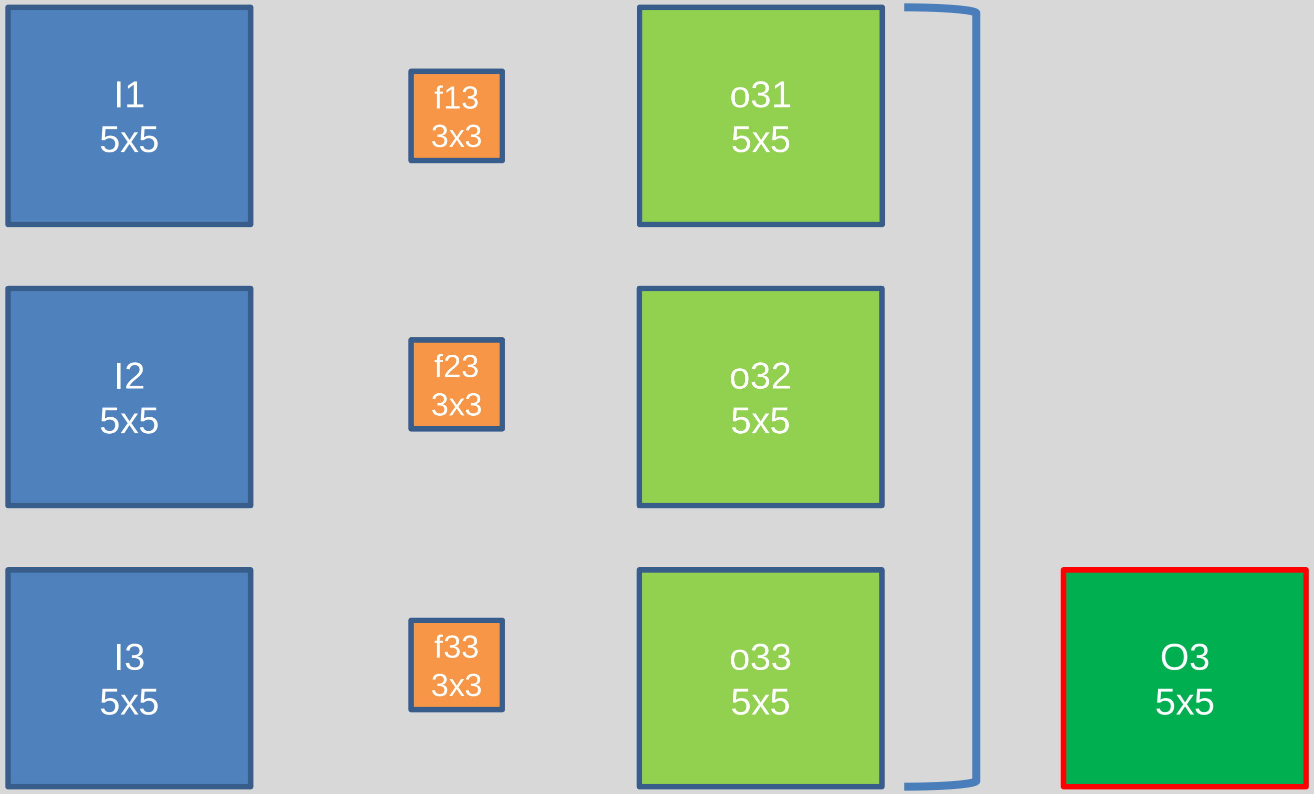
Convolutional layer



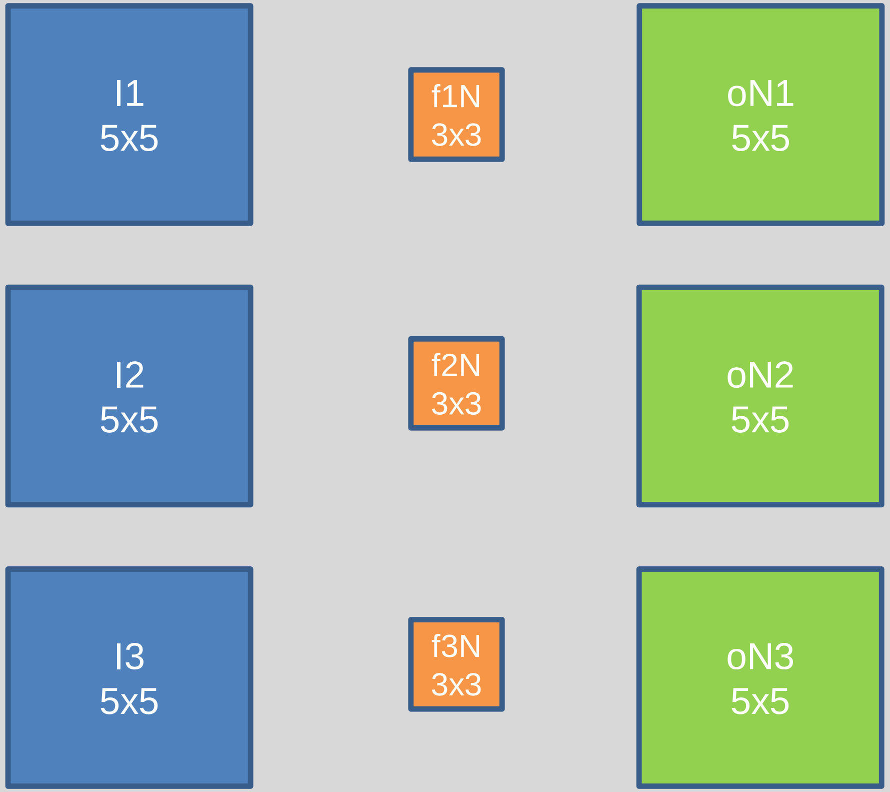
Convolutional layer



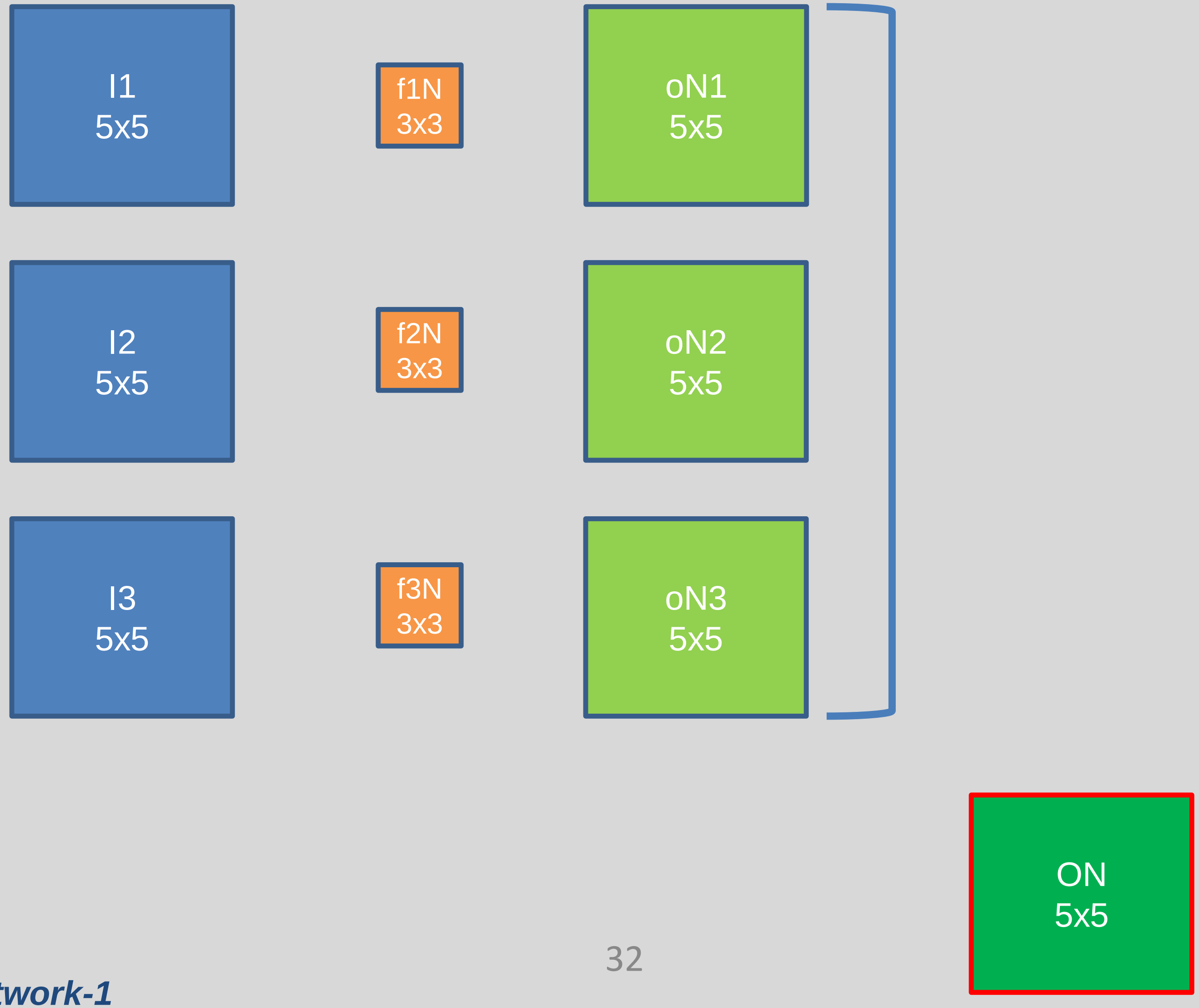
Convolutional layer



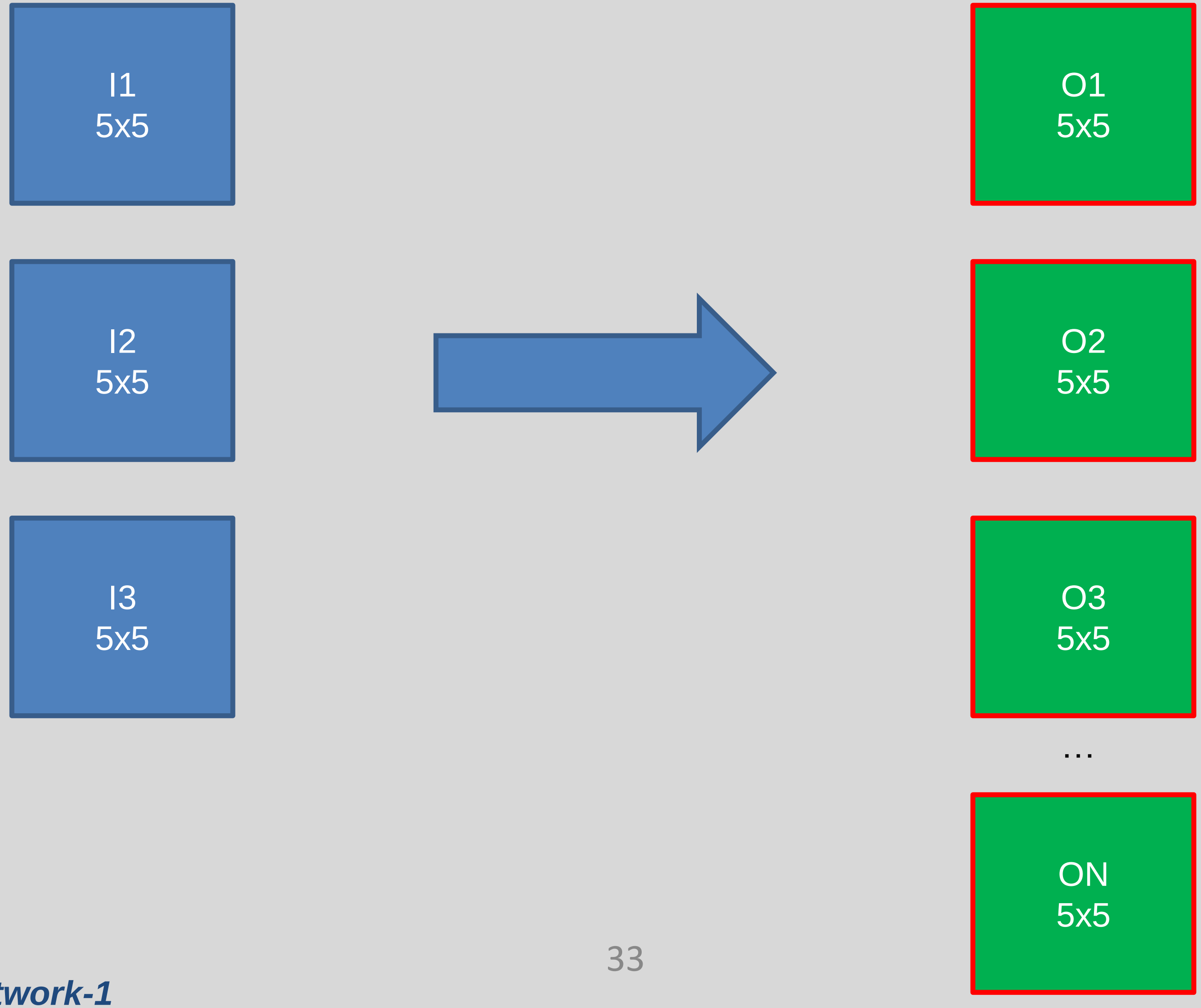
Convolutional layer



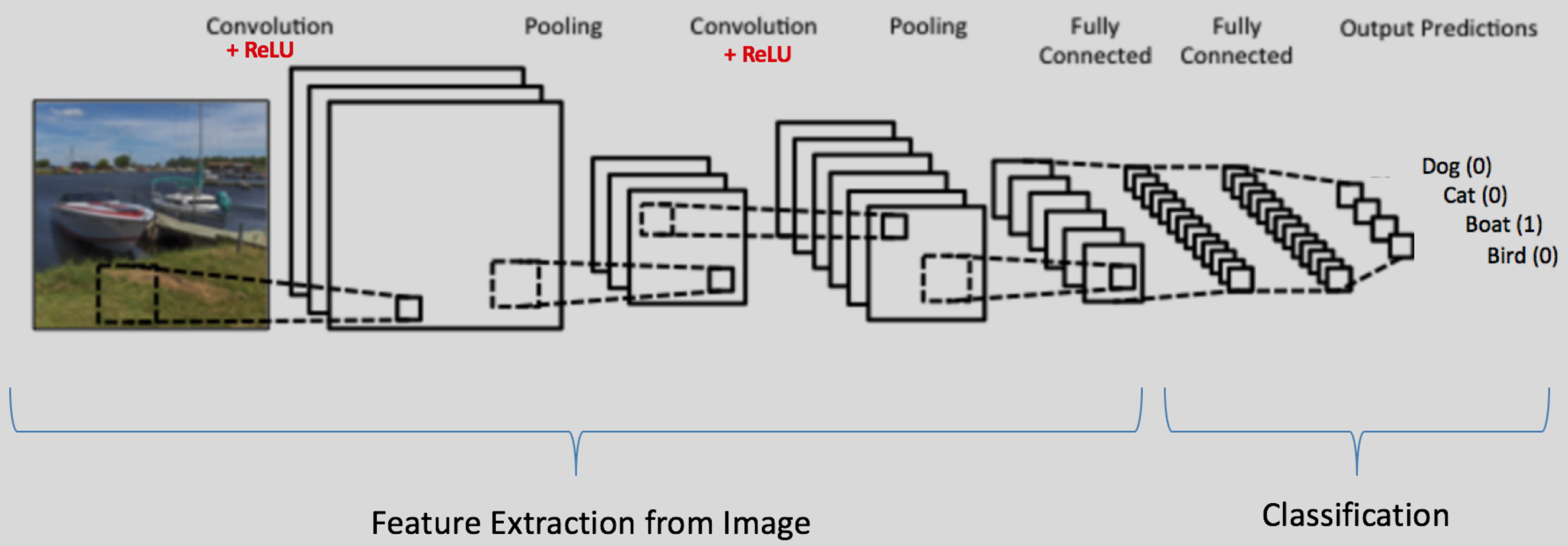
Convolutional layer



Convolutional layer



Convolutional neural network



Convolutional layer in PyTorch

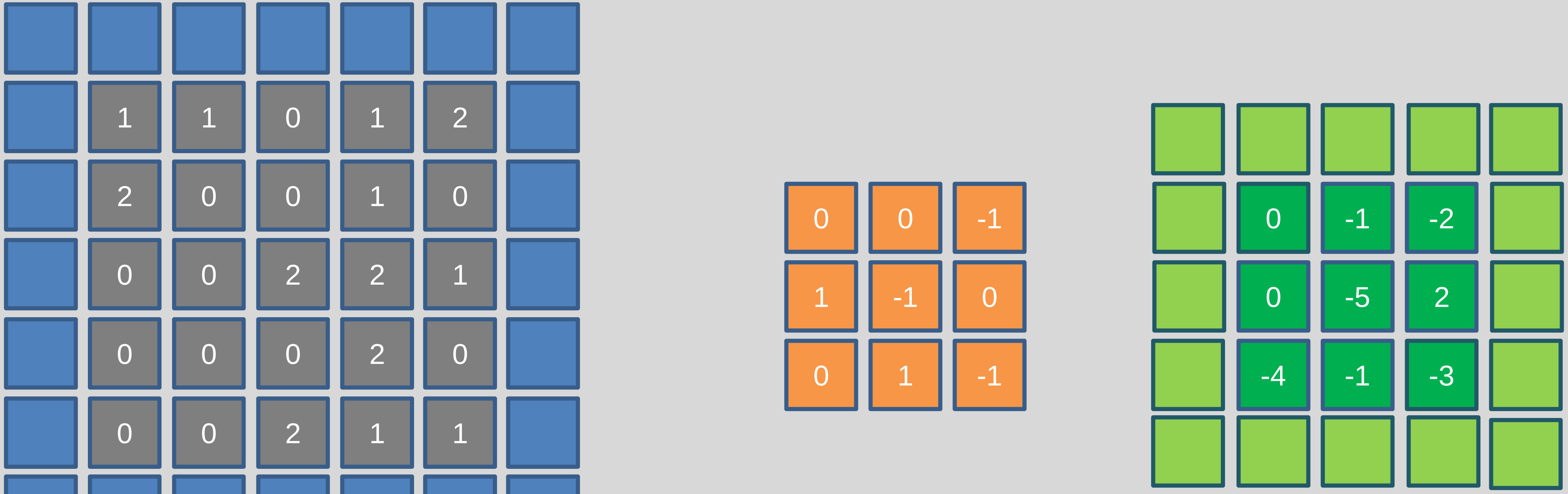


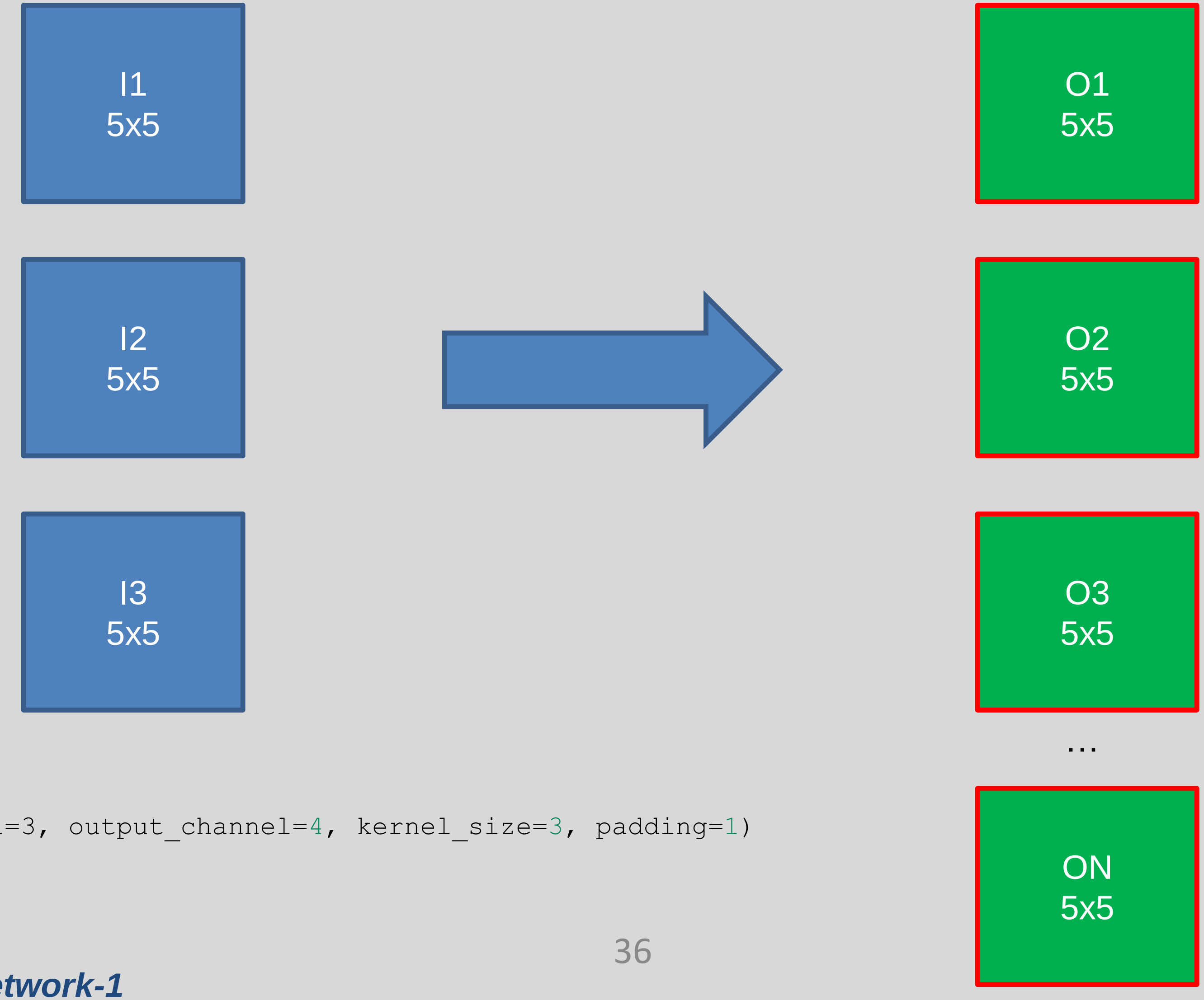
Image (5x5)

Filter kernel (3x3)

Output feature (5x5)

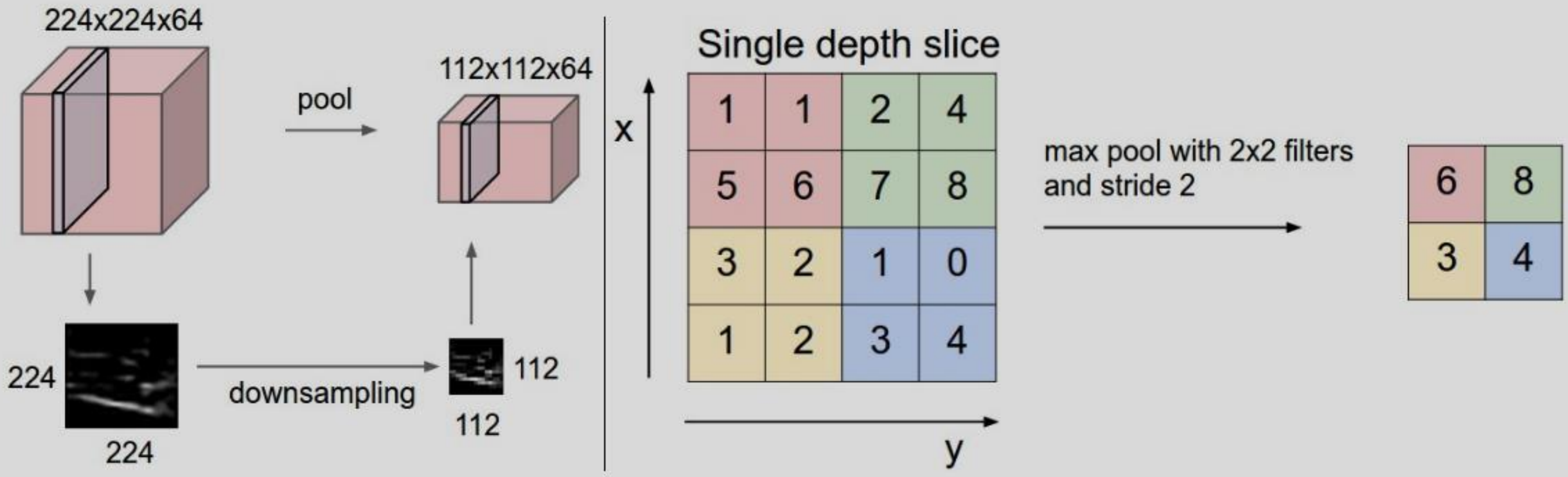
```
torch.nn.Conv2d(input_channel=1, output_channel=1, kernel_size=3, padding=1)
```

Convolutional layer

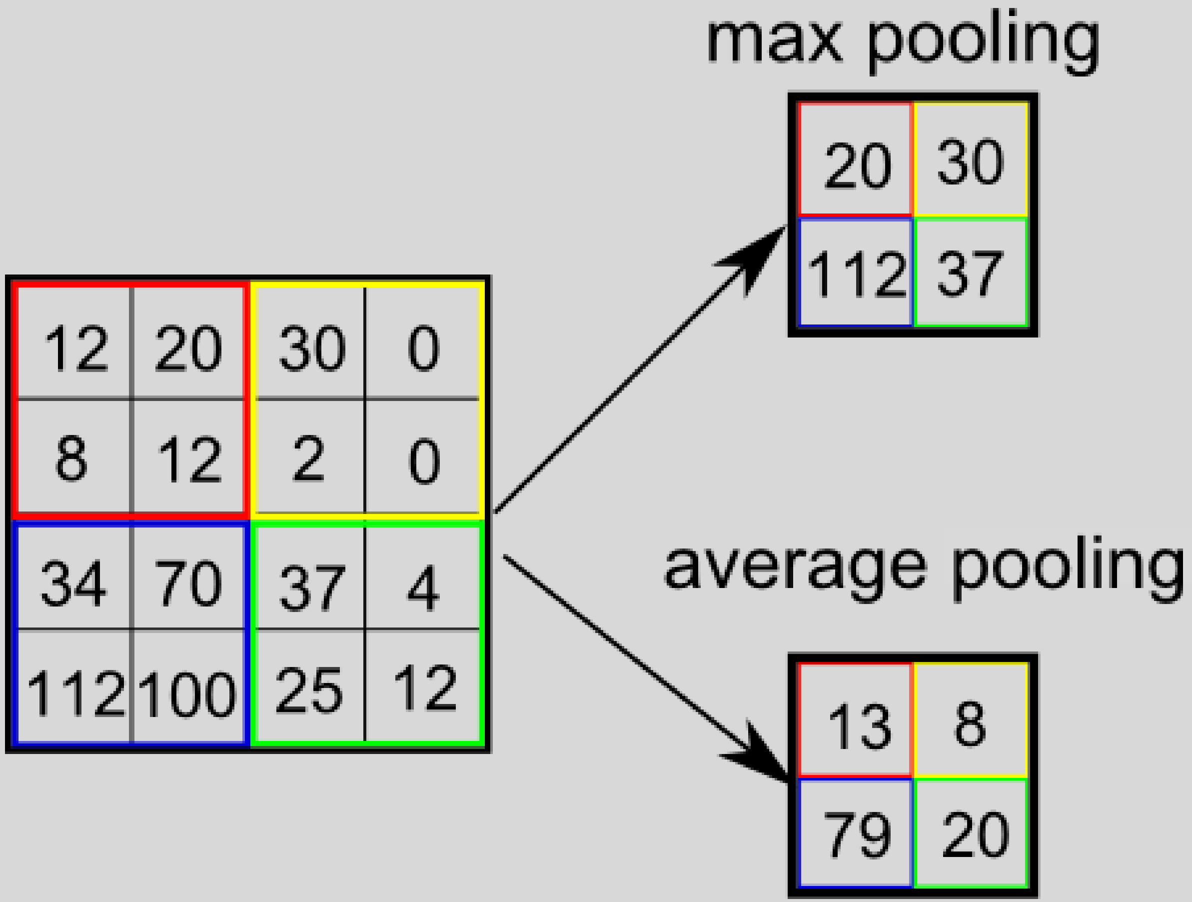



```
torch.nn.Conv2d(input_channel=3, output_channel=4, kernel_size=3, padding=1)
```

Pooling


224x224x64
pool
112x112x64
224
downsampling
112
224
112
Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

Pooling



torch.nn.MaxPool2d

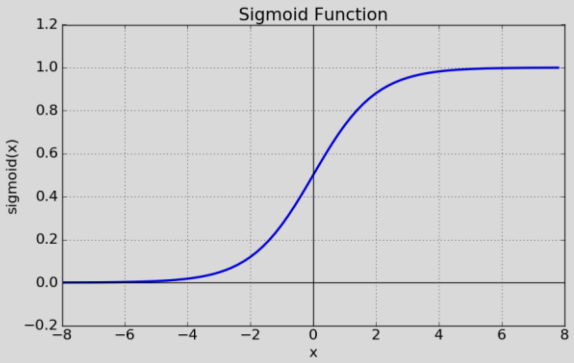
torch.nn.AvgPool2d

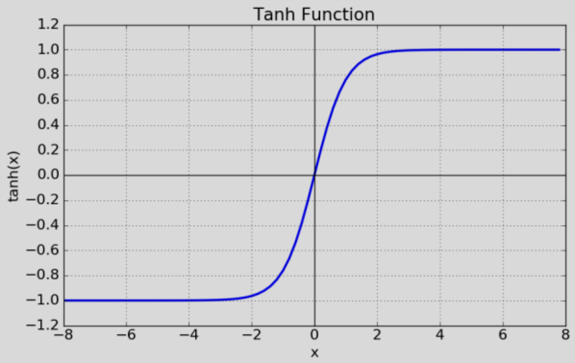
Activation layer

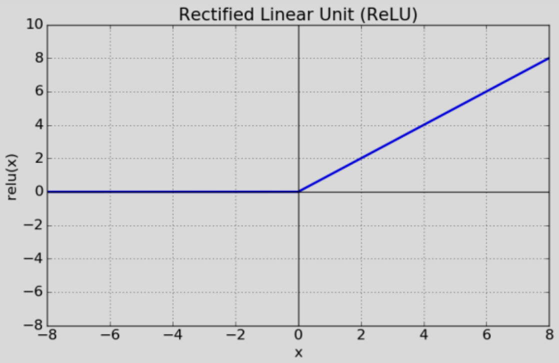
$$f(x) = \frac{1}{1+e^{-x}}$$

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$f(x) = max(0, x)$$





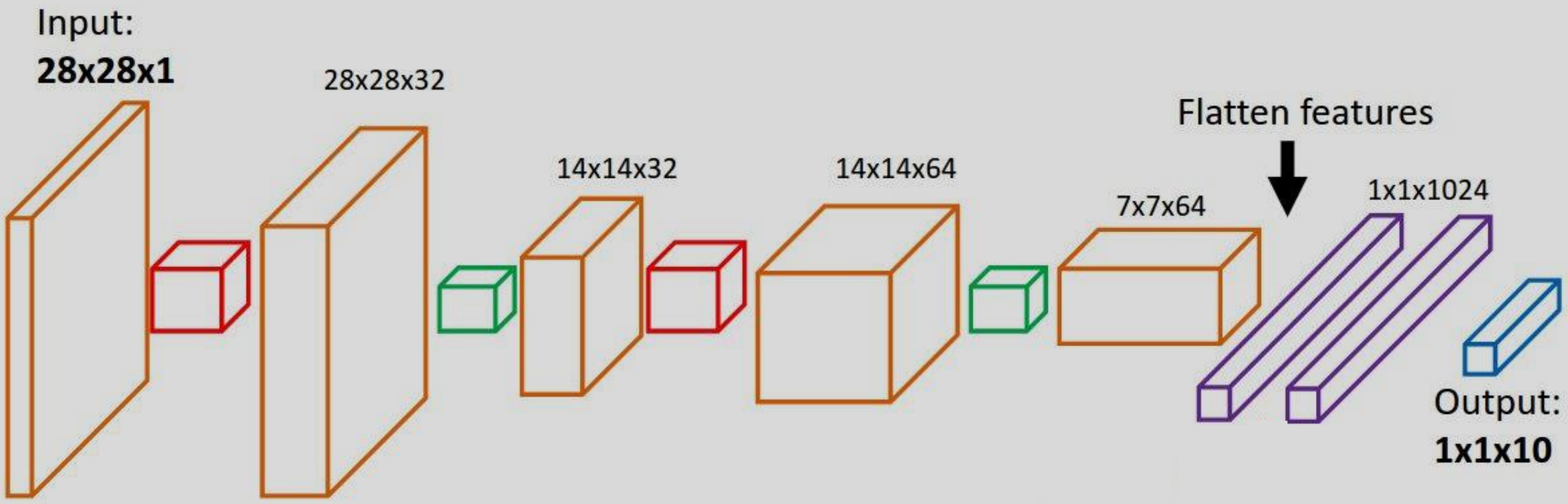


torch.nn.Sigmoid()

torch.nn.Tanh()

torch.nn.ReLU()

Overall CNN architecture

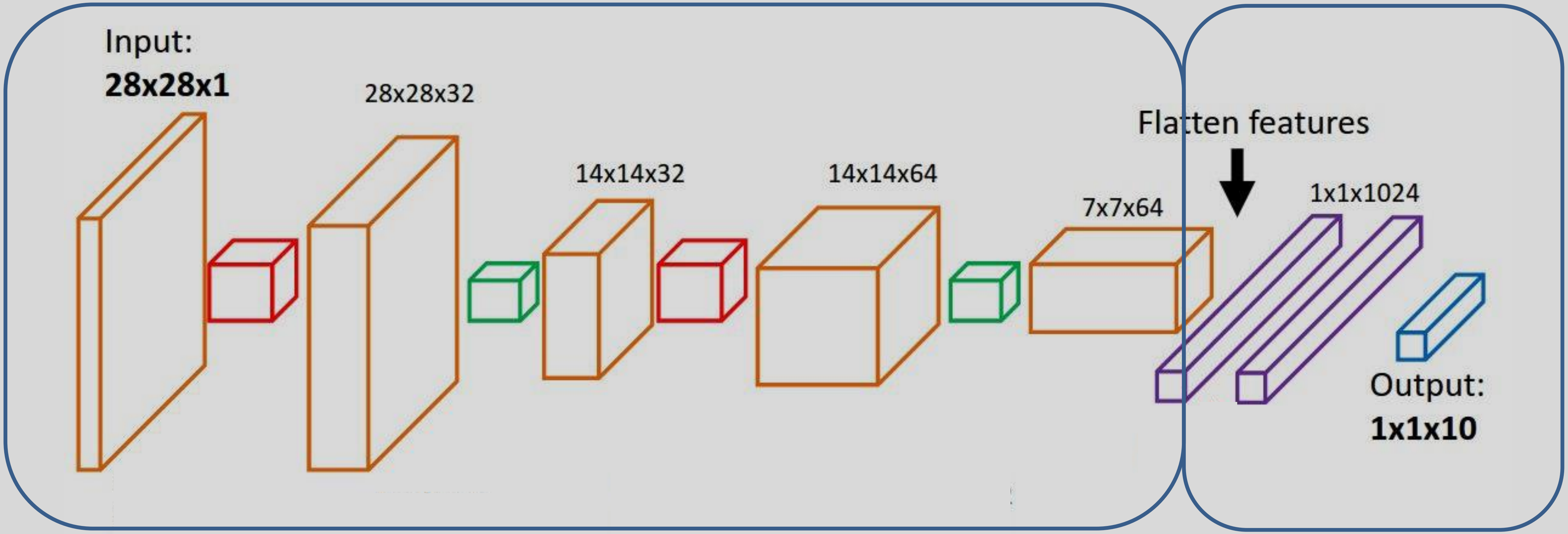


Combination of differentiable layers → Differentiable architecture!

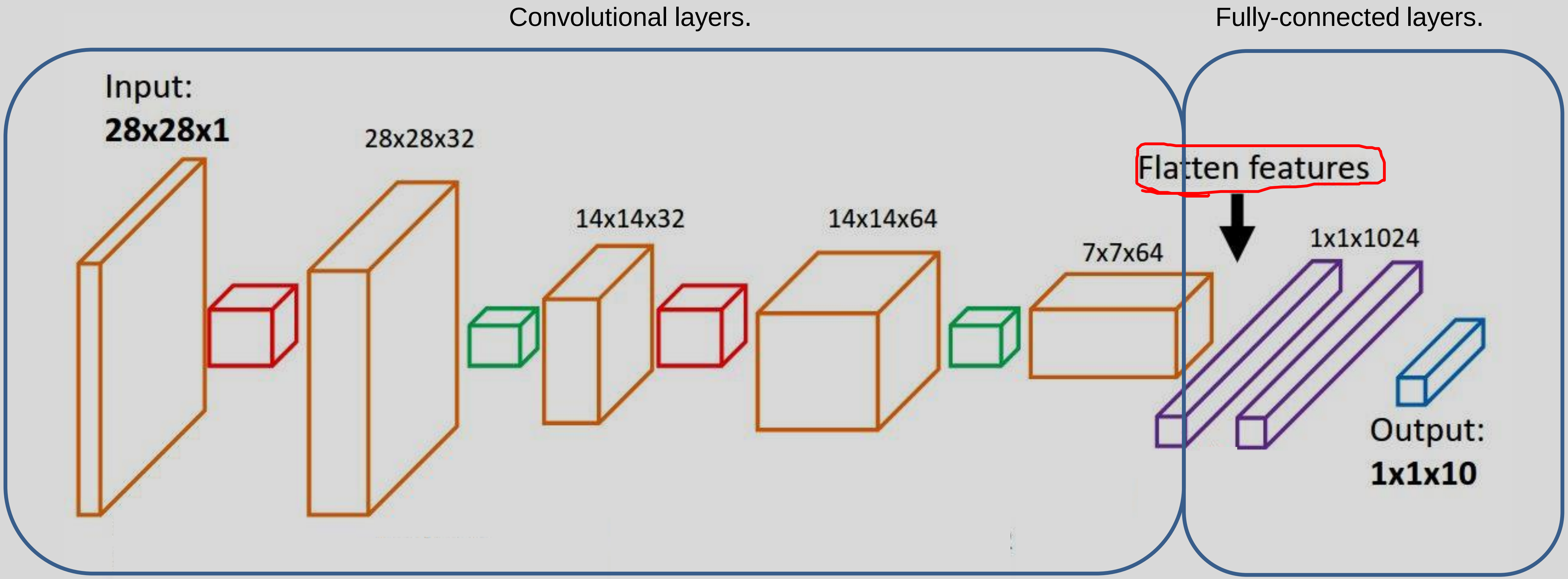
Overall CNN architecture

Convolutional layers.

Fully-connected layers.



Overall CNN architecture



Similar to feature encoding stage of conventional machine learning (ML) e.g. BOW representation.

Similar to classification stage of conventional ML e.g. SVM.

CNN implementation in PyTorch

```
import torch

class MyCNN(torch.nn.Module):
  def __init__(self):
    super().__init__()



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

Called when your network is initialized.

Called when the forward pass is performed on input x.

CNN implementation in PyTorch

```
import torch
import torch.nn

class MyCNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.layer = nn.Sequential(
        nn.Conv2d(1, 16, 5),
        nn.ReLU(),
        nn.Conv2d(16, 32, 5),
        nn.MaxPool2d(2, 2)
        nn.Conv2d(32, 64, 5)
        nn.ReLU()
        nn.MaxPool2d(2,2)
    )
    self.fc_layer = nn.Sequential(
        nn.Linear(64*3*3, 10)
        nn.ReLu()
        nn.Linear(100, 10)
    )

  def forward(self, x):
    out = self.layer(x)
    out = out.view(batch_size, -1)
    out = self.fc_layer(out)

    return out
```

Convolutional layers are generated.

Fully connected layers are generated.

Forward pass is defined.

Backward is automatically performed when calling loss.backward()

CNN implementation in PyTorch

```
import torch
import torch.nn

class MyCNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.layer = nn.Sequential(
        nn.Conv2d(1, 16, 5, padding=2),
        nn.ReLU(),
        nn.Conv2d(16, 32, 5),
        nn.MaxPool2d(2, 2)
        nn.Conv2d(32, 64, 5)
        nn.ReLU()
        nn.MaxPool2d(2,2)
    )
    self.fc_layer = nn.Sequential(
        nn.Linear(64*3*3, 10)
        nn.ReLu()
        nn.Linear(100, 10)
    )

  def forward(self, x):
    out = self.layer(x)
    out.out.view(batch_size, -1)
    out = self.fc_layer(out)

    return out
```

```
import torch

net = MyCNN()

loss_func = torch.nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

losses = []

for i in range(num_epoch):
  optimizer.zero_grad()

  output = net(x)

  loss = loss_func(output, y)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())
```

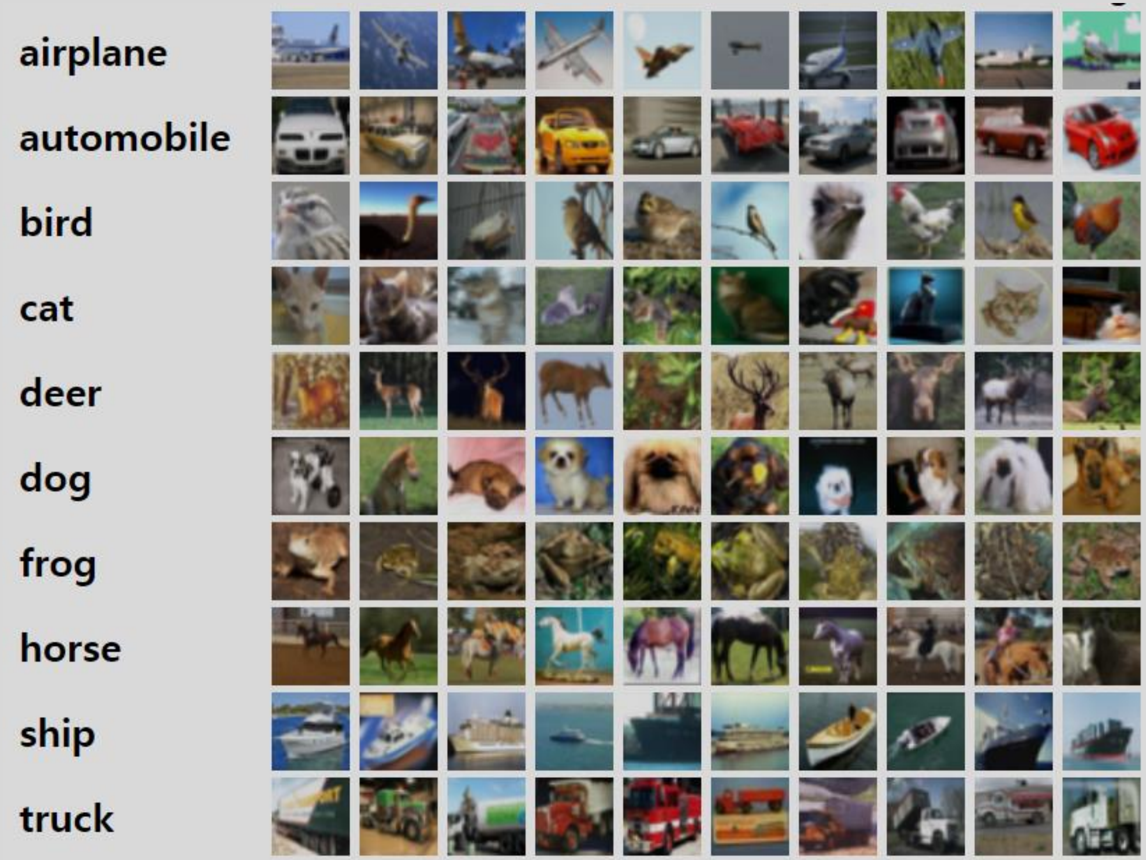
CNN implementation in PyTorch

```
import torch
import torch.nn

class MyCNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.layer = nn.Sequential(
        nn.Conv2d(1, 16, 5),
        nn.ReLU(),
        nn.Conv2d(16, 32, 5),
        nn.MaxPool2d(2, 2)
        nn.Conv2d(32, 64, 5)
        nn.ReLU()
        nn.MaxPool2d(2,2)
    )
    self.fc_layer = nn.Sequential(
        nn.Linear(64*3*3, 10)
        nn.ReLu()
        nn.Linear(100, 10)
    )

  def forward(self, x):
    out = self.layer(x)
    out = out.view*(batch_size, -1)
    out = self.fc_layer(out)

    return out
```

```
import torch

net = MyCNN()

loss_func = torch.nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

losses = []

for i in range(num_epoch):
  optimizer.zero_grad()

  output = net(x)

  loss = loss_func(output, y)

  loss.backward()
  optimizer.step()

  losses.append(loss.item())
```

Image Classification task

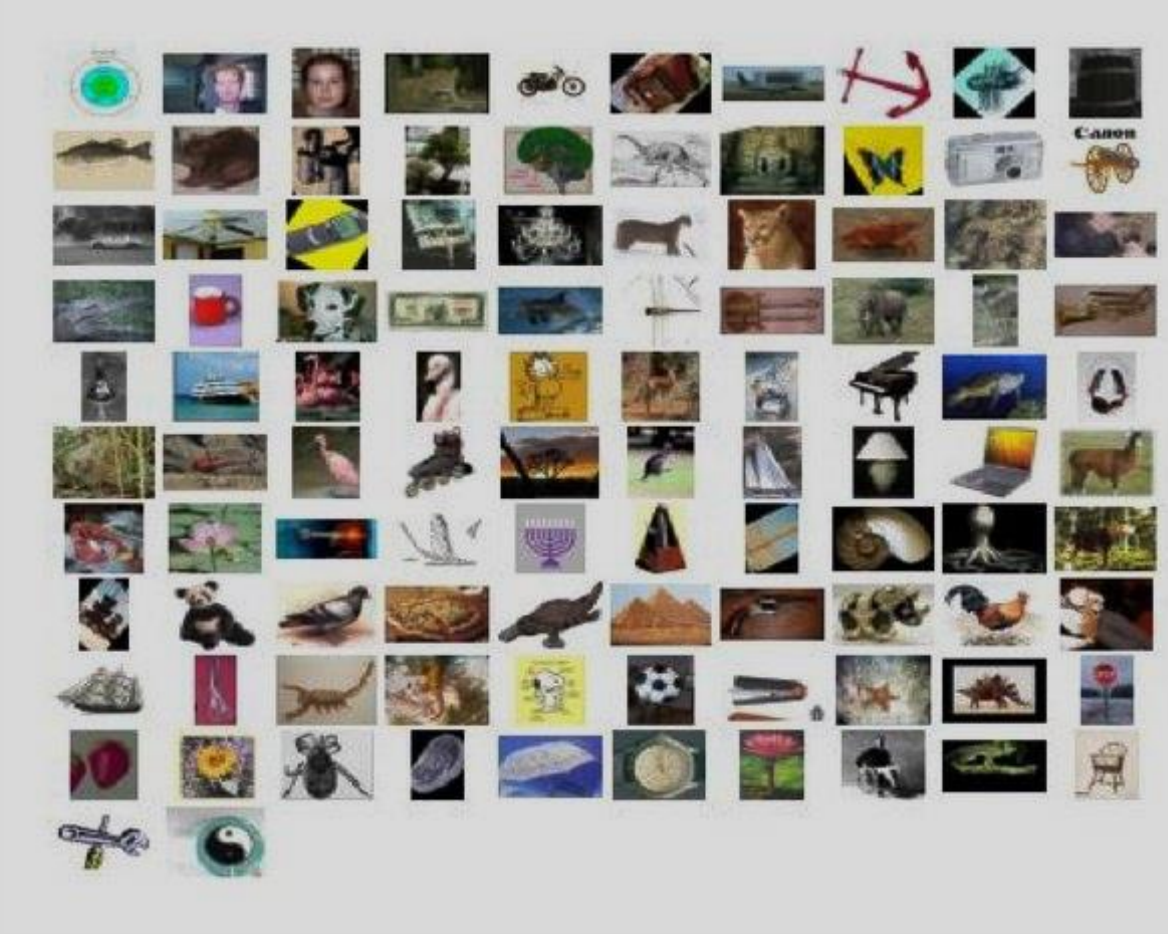
Cifar 10 dataset (10 classes)



MNIST dataset (10 classes)

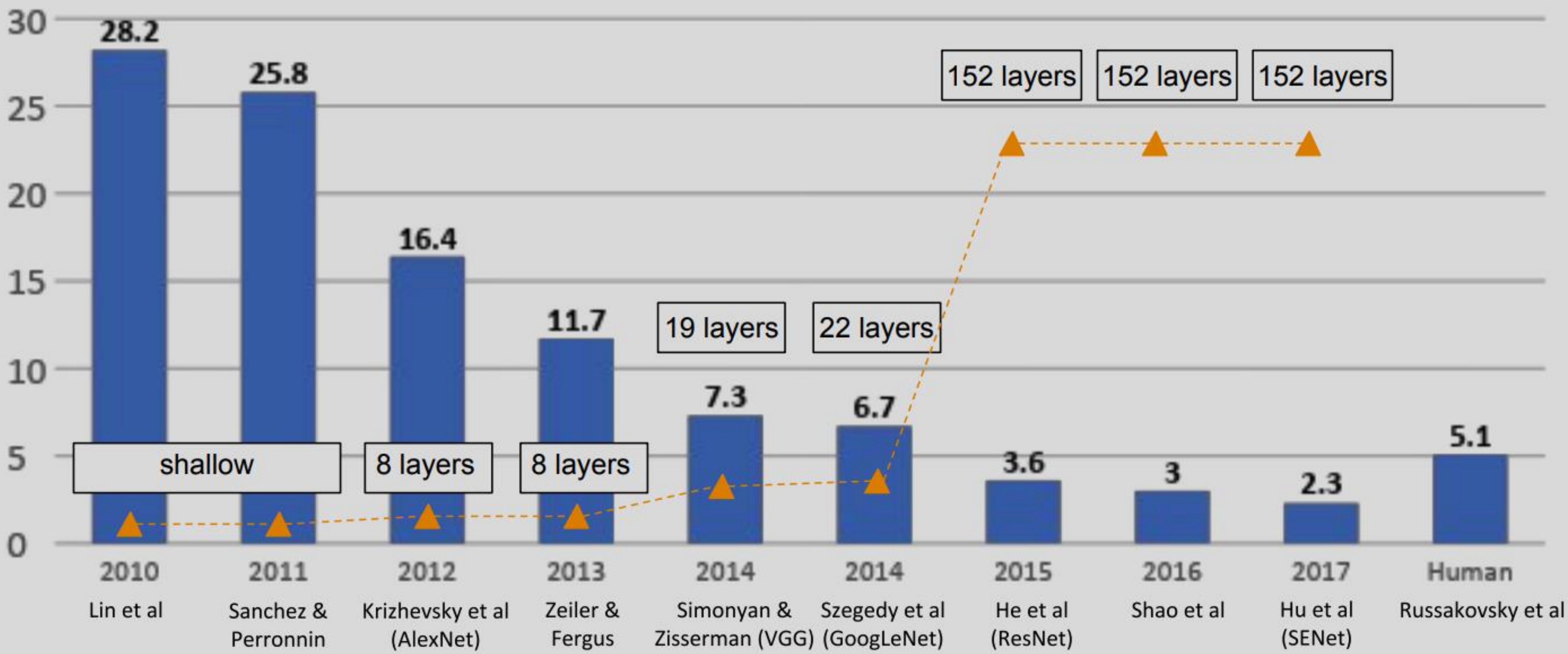
Image Classification task

Caltech 101 dataset (101 classes)



ImageNet dataset (1000 classes)

Image Classification task



Regression Loss

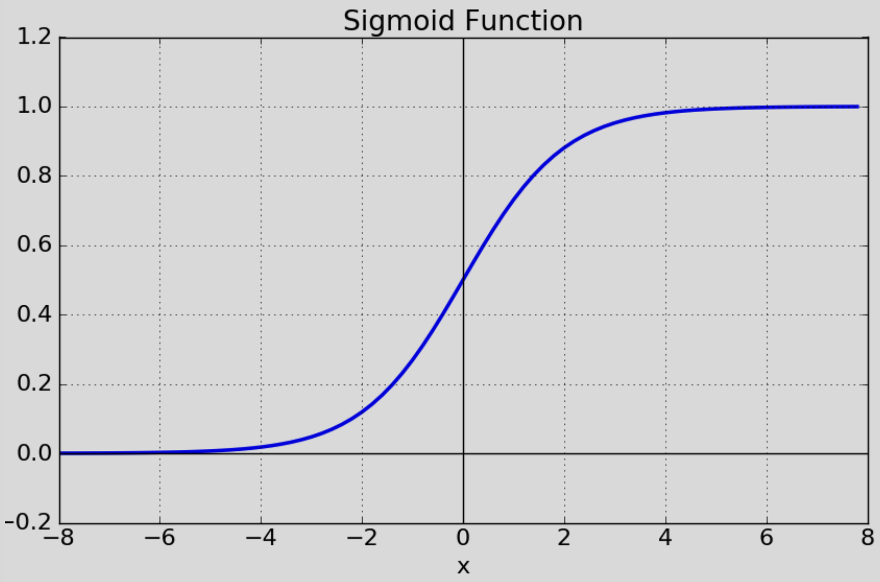
```python
import torch
import torch.nn as nn

loss1 = nn.MSELoss()
loss2 = nn.L1Loss()
```

$$Loss1 = ||x - y||_2^2$$
$$Loss2 = ||x - y||_1$$

Logistic Regression

Called as the regression, but actually performs the **binary classification**!

$$z = \frac{1}{1 + e^{-\mathbf{w}x+b}}$$
$$= \sigma(-\mathbf{w}x + b)$$



Logistic Regression

$$Loss = \begin{cases} -\ln z_n, y_n = 1 \\ -\ln(1 - z_n), y_n = 0 \end{cases}$$

$$Loss = -\sum_n y_n \ln z_n + (1 - y_n) \ln(1 - z_n)$$

Logistic Regression

```
import torch
import torch.nn as nn
from sklearn.datasets import load_iris
iris = load_iris()

X = iris.data[:100]
y = iris.target[:100]
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)

net = nn.Linear(4, 1)
loss_fn = nn.BCELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)

losses = []

for epoch in range(100):
  optimizer.zero_grad()

  h = net(X)
  prob = nn.functional.sigmoid(h)

  loss = loss_fn(prob, y)
  loss.backward()

  optimizer.step()
  losses.append(loss.item())
```

Logistic Regression

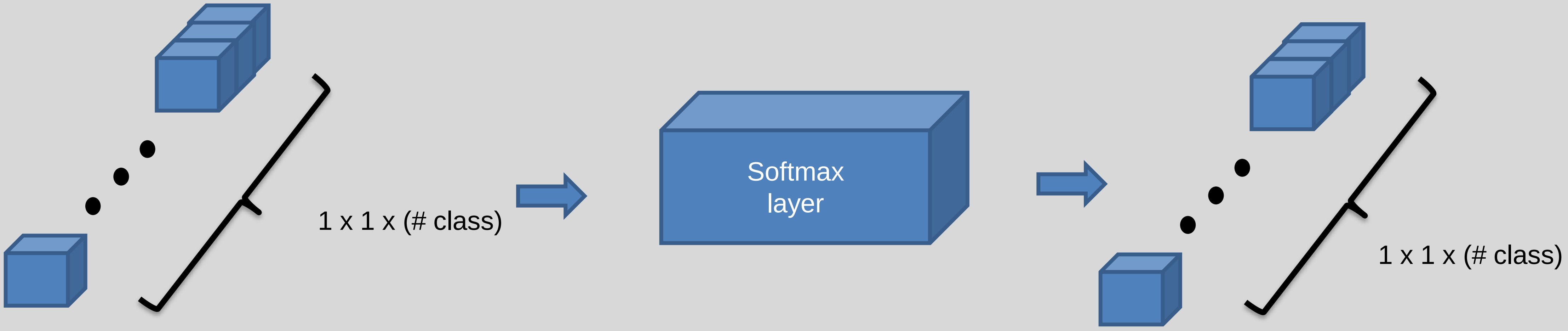
```
import torch
import torch.nn as nn
from sklearn.datasets import load_iris
iris = load_iris()

X = iris.data[:100]
y = iris.target[:100]
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)

net = nn.Linear(4, 1)
loss_fn = nn.BCEWithLogitsLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)

losses = []

for epoch in range(100):
  optimizer.zero_grad()

  h = net(X)

  loss = loss_fn(h.view_as(y), y)
  loss.backward()

  optimizer.step()
  losses.append(loss.item())
```

Logistic Regression

```
import torch
import torch.nn as nn
from sklearn.datasets import load_iris
iris = load_iris()

X = iris.data[[1, 51]]
y = iris.target[[1, 51]]
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)

net = nn.Linear(4, 1)

h=net(X)
prob = nn.functional.sigmoid(h)

loss_fn = nn.BCELoss()
loss_fn2 = nn.BCEWithLogitsLoss()

loss1 = loss_fn(prob, y)
loss2 = loss_fn2(h.view_as(y), y)

print(loss1, loss2)
```

```
tensor(1.2881, grad_fn=<BinaryCrossEntropyBackward>) tensor(1.2881, grad_fn=<BinaryCrossEntropyWithLogitsBackward>)
```

Softmax for multi-class classification



$$Softmax(y_i) = \frac{\exp(y_i)}{\sum_j \exp(y_j)}$$

The vector is L1-normalized. → It could mean probability for semantic classes.

Cross-entropy Loss

$$H(p,q) = -\sum_x p(x)\log q(x)$$

$$= \underbrace{-\sum_x p(x)\log p(x)}_{\text{Constant}} + \underbrace{\sum_x p(x)\log\frac{p(x)}{q(x)}}_{\text{KL divergence}}$$

Cross-entropy Loss

$$D_{KL}(p||q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

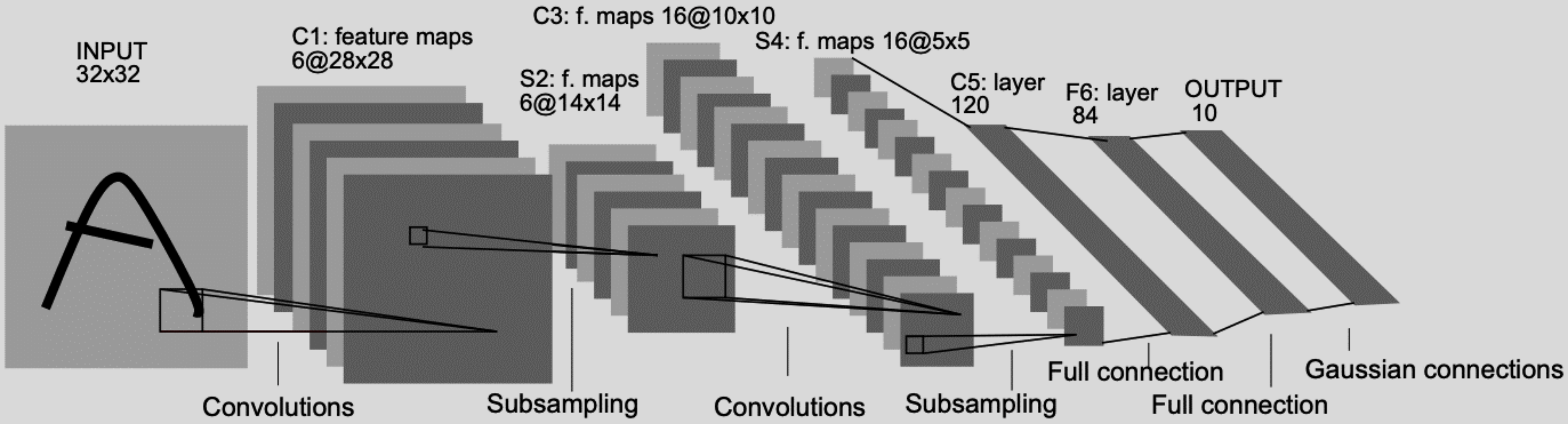
The value becomes 0 when p(x) = q(x). It is the minimum.

Cross-entropy Loss

```python
import torch
import torch.nn as nn

loss = nn.CrossEntropyLoss()

input = torch.randn(3, 5, requires_grad=True)
target = torch.empty(3, dtype=torch.long).random_(5)

output = loss(input, target)

print(input)
print(target)
print(output)
```

```
tensor([[-1.8600, -1.1599,  0.0525, -1.9408, -1.4070],
        [ 0.6205, -1.8228,  2.1522, -2.5549, -0.1288],
        [ 0.1473, -1.5891,  1.0388, -0.6910,  0.4188]], requires_grad=True)
tensor([2, 1, 0])
tensor(2.1822, grad_fn=<NllLossBackward>)
```

LeNet



[Yann Lecun 1998.]

Implementing LeNet

```
import numpy as np

import torch
import torch.nn as nn

from torch.utils.data import DataLoader
from torchvision import datasets, transforms

import matplotlib.pyplot as plt

LEARNING_RATE = 0.001
BATCH_SIZE = 32
N_EPOCHS = 100

IMG_SIZE = 32
N_CLASSES = 10
```

LeNet

```
class LeNet5(nn.Module):

    def __init__(self, n_classes):
        super(LeNet5, self).__init__()

        self.feature_extractor = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5, stride=1),
            nn.Tanh()
        )

        self.classifier = nn.Sequential(
            nn.Linear(in_features=120, out_features=84),
            nn.Tanh(),
            nn.Linear(in_features=84, out_features=n_classes),
        )


    def forward(self, x):
        x = self.feature_extractor(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

Data Loader

```
trans = transforms.Compose([transforms.Resize((32, 32)), transforms.ToTensor()])

train_dataset = datasets.MNIST(root = 'mnist_data', train=True, transform=trans, download=True)
test_dataset = datasets.MNIST(root = 'mnist_data', train=False, transform=trans)

train_loader = DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

Frequently used Datasets are readily available!

https://pytorch.org/docs/stable/torchvision/datasets.html

Data Loader

```
trans = transforms.Compose([transforms.Resize((32, 32)), transforms.ToTensor()])

train_dataset = datasets.MNIST(root = 'mnist_data', train=True, transform=trans, download=True)
test_dataset = datasets.MNIST(root = 'mnist_data', train=False, transform=trans)

train_loader = DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

You can also define your own dataset for data loader.

```
from torch.utils.data import Dataset
from torch.utils.data import DataLoader

class CustomDataset(Dataset):

   def __init__(self):
      self.x_data = [[73, 80, 75], [93, 88, 93], [89, 91, 90], [96, 98, 100], [73, 66, 70]]
      self.y_data = [[152], [185], [180], [196], [142]]

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

   def __getitem__(self, idx):
      x = torch.FloatTensor(self.x_data[idx])
      y = torch.FloatTensor(self.y_data[idx])
      return x, y
```

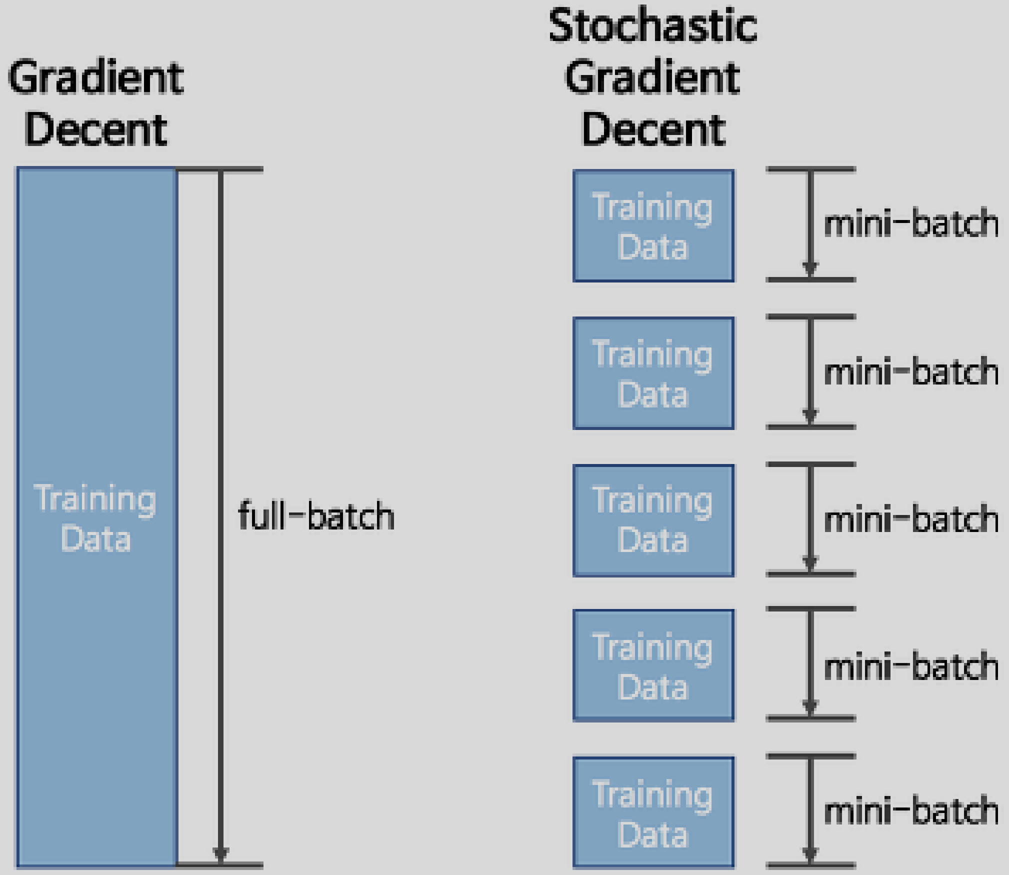
```
for X, y_true in train_loader:
```

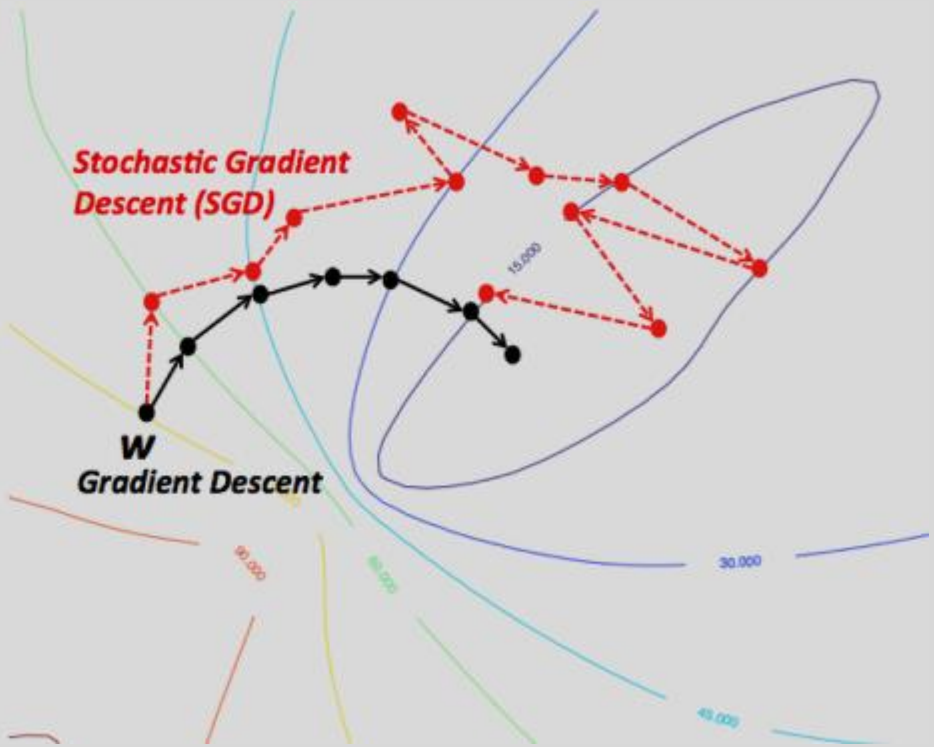
Stochastic Gradient Descent

- Gradient descent
  Calculate for all data (takes large amount of time).
  Go 1 optimal step.

- Stochastic gradient descent
  Calculate gradients for partial data (takes small amount of time).
  Go many non-globally-optimal steps, but converges.

Stochastic Gradient Descent





Implementing LeNet

```
def train(train_loader, model, criterion, optimizer):

    model.train()
    train_loss = 0
    correct = 0

    for X, y_true in train_loader:

        optimizer.zero_grad()

        y_hat = model(X)
        loss = criterion(y_hat, y_true)

        train_loss += loss.item()
        pred = y_hat.argmax(dim=1, keepdim=True)
        correct += pred.eq(y_true.view_as(pred)).sum().item()

        loss.backward()
        optimizer.step()

    epoch_loss = train_loss / len(train_loader.dataset)
    acc = correct / len(train_loader.dataset)

    return model, optimizer, epoch_loss, acc
```

Training flag .train()

Same as before.

Implementing LeNet

```
def test(test_loader, model, criterion):

    model.eval()
    test_loss = 0
    correct = 0

    for X, y_true in test_loader:

        y_hat = model(X)
        loss = criterion(y_hat, y_true)

        test_loss += loss.item()
        pred = y_hat.argmax(dim=1, keepdim=True)
        correct += pred.eq(y_true.view_as(pred)).sum().item()

    epoch_loss = test_loss / len(test_loader.dataset)
    acc = correct / len(test_loader.dataset)

    return model, epoch_loss, acc
```

Testing flag .eval()

Same as before, but without backward step.

Implementing LeNet

```
def training_loop(model, criterion, optimizer, train_loader, test_loader, epochs, print_every=1):

    train_losses = []
    test_losses = []

    for epoch in range(epochs):

        model, optimizer, train_loss, train_acc = train(train_loader, model, criterion, optimizer)
        train_losses.append(train_loss)

        with torch.no_grad():
            model, test_loss, test_acc = test(test_loader, model, criterion)
            test_losses.append(test_loss)

        if epoch % print_every == (print_every - 1):

            print(f'Epoch: {epoch}\t'
                  f'Train loss: {train_loss:.4f}\t'
                  f'Test loss: {test_loss:.4f}\t'
                  f'Train accuracy: {100 * train_acc:.2f}\t'
                  f'Test accuracy: {100 * test_acc:.2f}')

    return model, optimizer, (train_losses, test_losses)
```

No gradient calculation mode.

Implementing LeNet

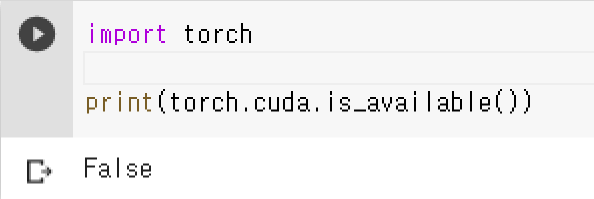
```
model = LeNet5(N_CLASSES)

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

model, optimizer, _ = training_loop(model, criterion, optimizer, train_loader, test_loader, N_EPOCHS)
```

Training using CPU

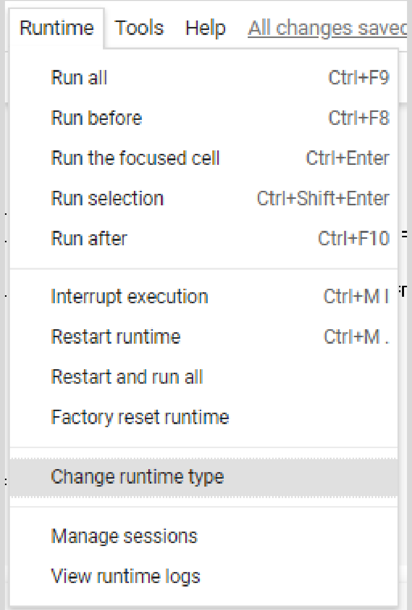
```
Epoch: 0 Train loss: 2.2817 Test loss: 2.2540 Train accuracy: 14.16 Test accuracy: 34.11
Epoch: 1 Train loss: 2.1948 Test loss: 2.0867 Train accuracy: 50.41 Test accuracy: 56.95
Epoch: 2 Train loss: 1.8551 Test loss: 1.5684 Train accuracy: 56.49 Test accuracy: 62.10
Epoch: 3 Train loss: 1.3052 Test loss: 1.0489 Train accuracy: 67.44 Test accuracy: 74.17
Epoch: 4 Train loss: 0.9017 Test loss: 0.7621 Train accuracy: 77.70 Test accuracy: 81.95
Epoch: 5 Train loss: 0.6930 Test loss: 0.6112 Train accuracy: 82.86 Test accuracy: 85.24
Epoch: 6 Train loss: 0.5783 Test loss: 0.5238 Train accuracy: 85.29 Test accuracy: 86.71
Epoch: 7 Train loss: 0.5090 Test loss: 0.4686 Train accuracy: 86.75 Test accuracy: 87.75
Epoch: 8 Train loss: 0.4633 Test loss: 0.4298 Train accuracy: 87.60 Test accuracy: 88.62
Epoch: 9 Train loss: 0.4306 Test loss: 0.4009 Train accuracy: 88.34 Test accuracy: 89.17
Epoch: 10 Train loss: 0.4056 Test loss: 0.3788 Train accuracy: 88.82 Test accuracy: 89.68
Epoch: 11 Train loss: 0.3855 Test loss: 0.3602 Train accuracy: 89.27 Test accuracy: 90.08
Epoch: 12 Train loss: 0.3686 Test loss: 0.3444 Train accuracy: 89.65 Test accuracy: 90.45
Epoch: 13 Train loss: 0.3540 Test loss: 0.3310 Train accuracy: 89.95 Test accuracy: 90.75
Epoch: 14 Train loss: 0.3407 Test loss: 0.3183 Train accuracy: 90.29 Test accuracy: 91.00
```

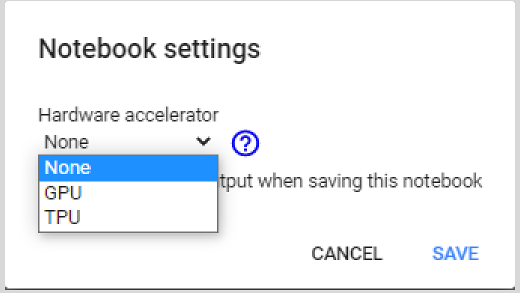
It takes about 15 minutes for training 15 epochs. – **slow!**

Using GPUs in Colab

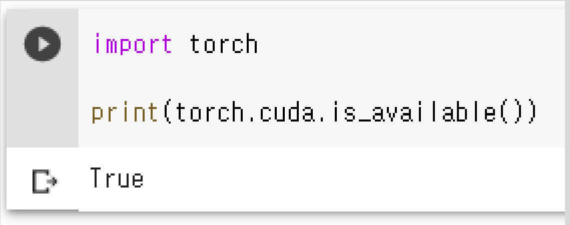


Colab does not support GPUs by default.





Select GPU.



Colab now supports GPUs!

Implementing LeNet

```
def train(train_loader, model, criterion, optimizer, device):

    model.train()
    train_loss = 0
    correct = 0

    for X, y_true in train_loader:
        optimizer.zero_grad()

        X = X.to(device)
        y_true = y_true.to(device)

        y_hat = model(X)
        loss = criterion(y_hat, y_true)

        train_loss += loss.item()
        pred = y_hat.argmax(dim=1, keepdim=True)
        correct += pred.eq(y_true.view_as(pred)).sum().item()

        loss.backward()
        optimizer.step()

    epoch_loss = train_loss / len(train_loader.dataset)
    acc = correct / len(train_loader.dataset)

    return model, optimizer, epoch_loss, acc
```

Implementing LeNet

```
def test(test_loader, model, criterion, device):

    model.eval()
    test_loss = 0
    correct = 0

    for X, y_true in test_loader:

        X = X.to(device)
        y_true = y_true.to(device)

        y_hat = model(X)
        loss = criterion(y_hat, y_true)

        test_loss += loss.item()
        pred = y_hat.argmax(dim=1, keepdim=True)
        correct += pred.eq(y_true.view_as(pred)).sum().item()

    epoch_loss = test_loss / len(test_loader.dataset)
    acc = correct / len(test_loader.dataset)

    return model, epoch_loss, acc
```

Implementing LeNet

```
def training_loop(model, criterion, optimizer, train_loader, test_loader, epochs, device, print_every=1):

    best_loss = 1e10
    train_losses = []
    test_losses = []

    for epoch in range(epochs):

        model, optimizer, train_loss, train_acc = train(train_loader, model, criterion, optimizer, device)
        train_losses.append(train_loss)

        with torch.no_grad():
            model, test_loss, test_acc = test(test_loader, model, criterion, device)
            test_losses.append(test_loss)

        if epoch % print_every == (print_every - 1):

            print(f'Epoch: {epoch}\t'
                  f'Train loss: {train_loss:.4f}\t'
                  f'Test loss: {test_loss:.4f}\t'
                  f'Train accuracy: {100 * train_acc:.2f}\t'
                  f'Test accuracy: {100 * test_acc:.2f}')

    return model, optimizer, (train_losses, test_losses)
```

Implementing LeNet

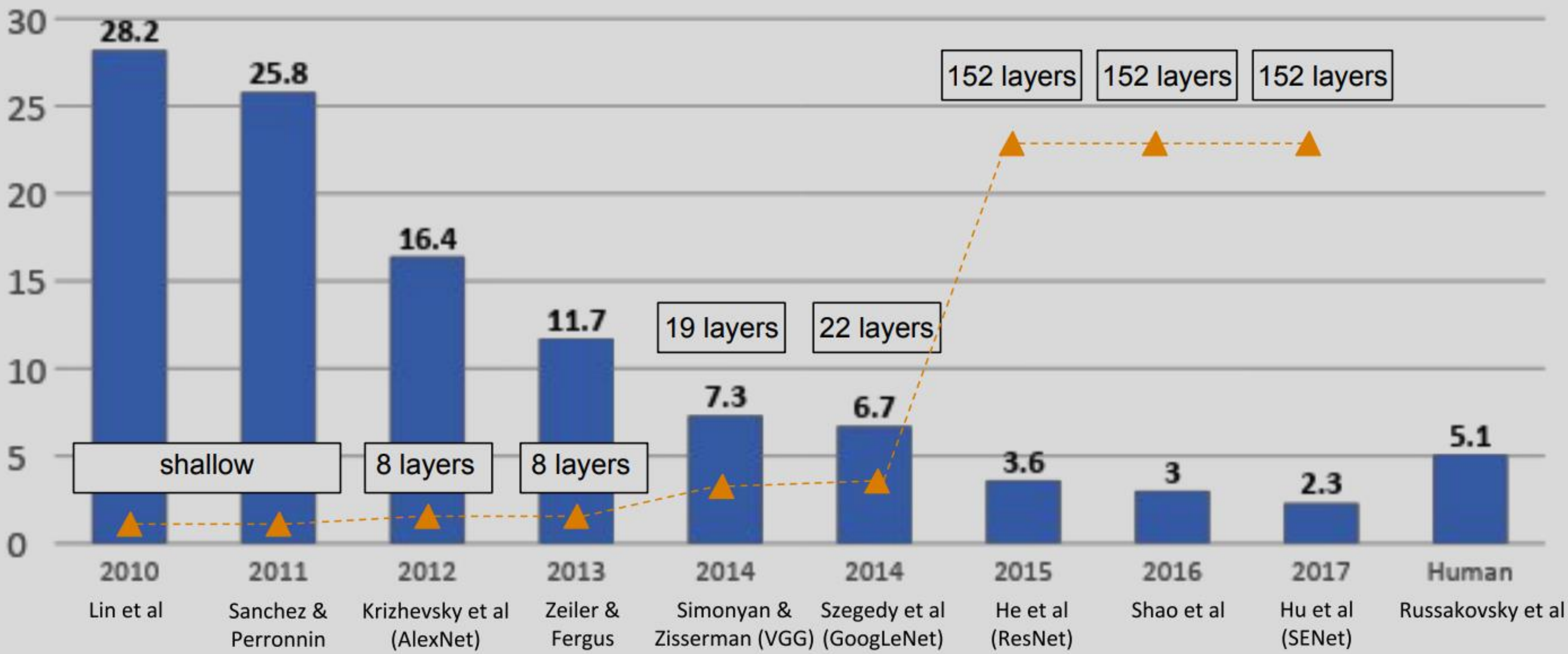
```
DEVICE = ‘cuda’ if torch.cuda.is_available() else ‘cpu’

model = LeNet5(N_CLASSES).to(DEVICE)

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

model, optimizer, _ = training_loop(model, criterion, optimizer, train_loader, test_loader, N_EPOCHS, DEVICE)
```

Training using GPU

```
Epoch: 0 Train loss: 2.2817 Test loss: 2.2540 Train accuracy: 14.16 Test accuracy: 34.11
Epoch: 1 Train loss: 2.1948 Test loss: 2.0867 Train accuracy: 50.41 Test accuracy: 56.95
Epoch: 2 Train loss: 1.8551 Test loss: 1.5684 Train accuracy: 56.49 Test accuracy: 62.10
Epoch: 3 Train loss: 1.3052 Test loss: 1.0489 Train accuracy: 67.44 Test accuracy: 74.17
Epoch: 4 Train loss: 0.9017 Test loss: 0.7621 Train accuracy: 77.70 Test accuracy: 81.95
Epoch: 5 Train loss: 0.6930 Test loss: 0.6112 Train accuracy: 82.86 Test accuracy: 85.24
Epoch: 6 Train loss: 0.5783 Test loss: 0.5238 Train accuracy: 85.29 Test accuracy: 86.71
Epoch: 7 Train loss: 0.5090 Test loss: 0.4686 Train accuracy: 86.75 Test accuracy: 87.75
Epoch: 8 Train loss: 0.4633 Test loss: 0.4298 Train accuracy: 87.60 Test accuracy: 88.62
Epoch: 9 Train loss: 0.4306 Test loss: 0.4009 Train accuracy: 88.34 Test accuracy: 89.17
Epoch: 10 Train loss: 0.4056 Test loss: 0.3788 Train accuracy: 88.82 Test accuracy: 89.68
Epoch: 11 Train loss: 0.3855 Test loss: 0.3602 Train accuracy: 89.27 Test accuracy: 90.08
Epoch: 12 Train loss: 0.3686 Test loss: 0.3444 Train accuracy: 89.65 Test accuracy: 90.45
Epoch: 13 Train loss: 0.3540 Test loss: 0.3310 Train accuracy: 89.95 Test accuracy: 90.75
Epoch: 14 Train loss: 0.3407 Test loss: 0.3183 Train accuracy: 90.29 Test accuracy: 91.00
```

Same results but it only takes 1-2 mins. For 15 epochs. – **much faster!**

Next class: AlexNet, VGG, GoogLeNet, ResNet



Thank you!