Convolutional Neural Network

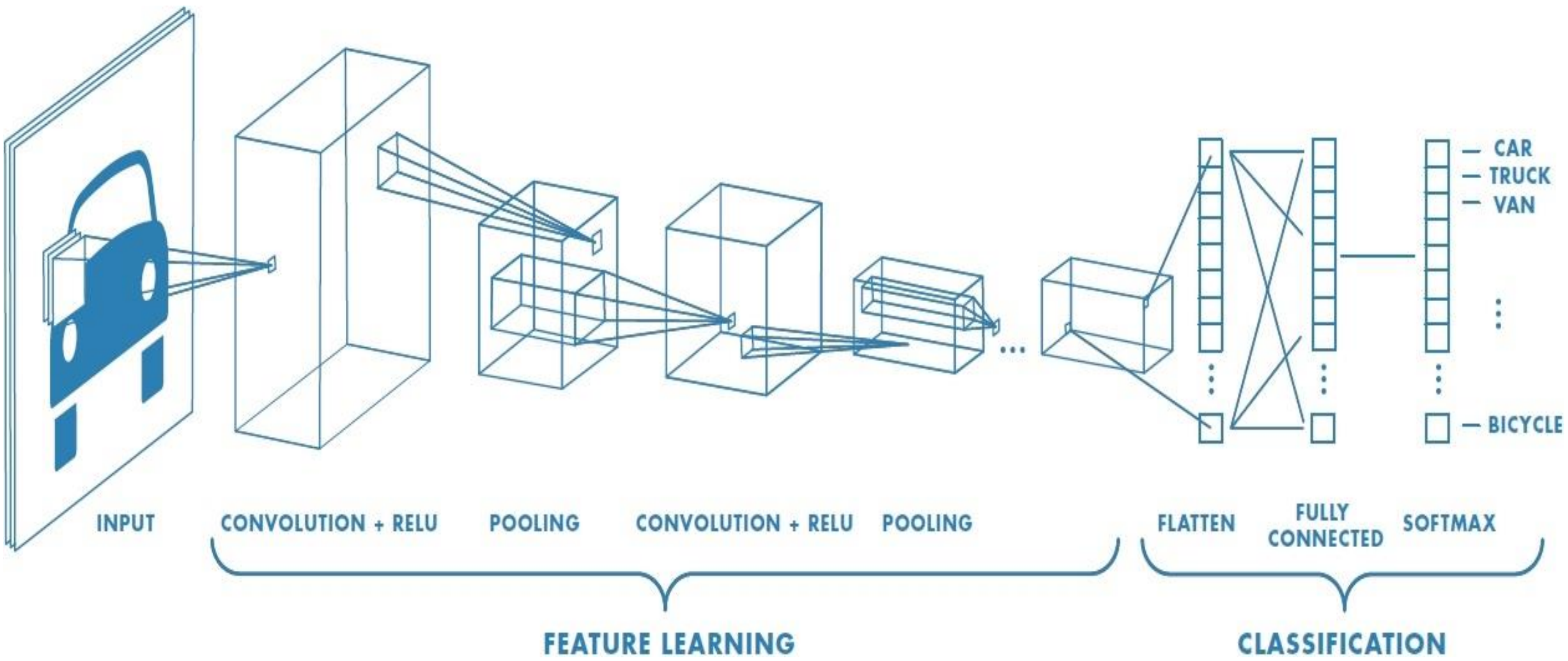
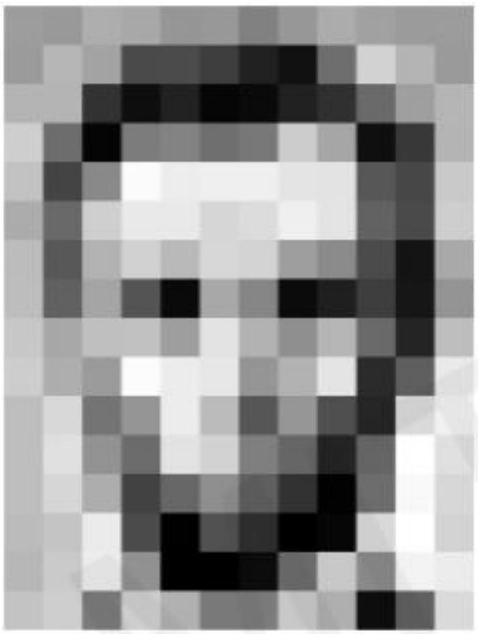


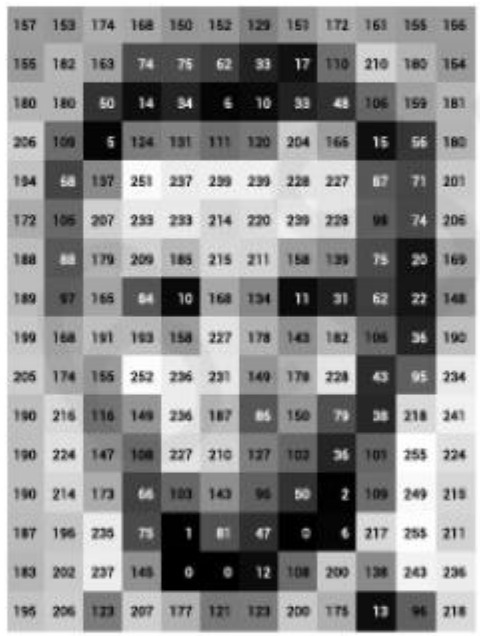
Image source

CNN

- CNN learn from images both:
  - which features to use (i.e., detects patterns)
  - how to classify them
- CNN can learn filters to detect patterns
  - Avoid the need for manual features engineering
- CNN uses Convolution, ReLU, Pooling
  - 1. Learn features in input image through convolution
  - 2. Introduce non-linearity through activation function
  - 3. Reduce dimensionality with pooling

Images are matrix of numbers!



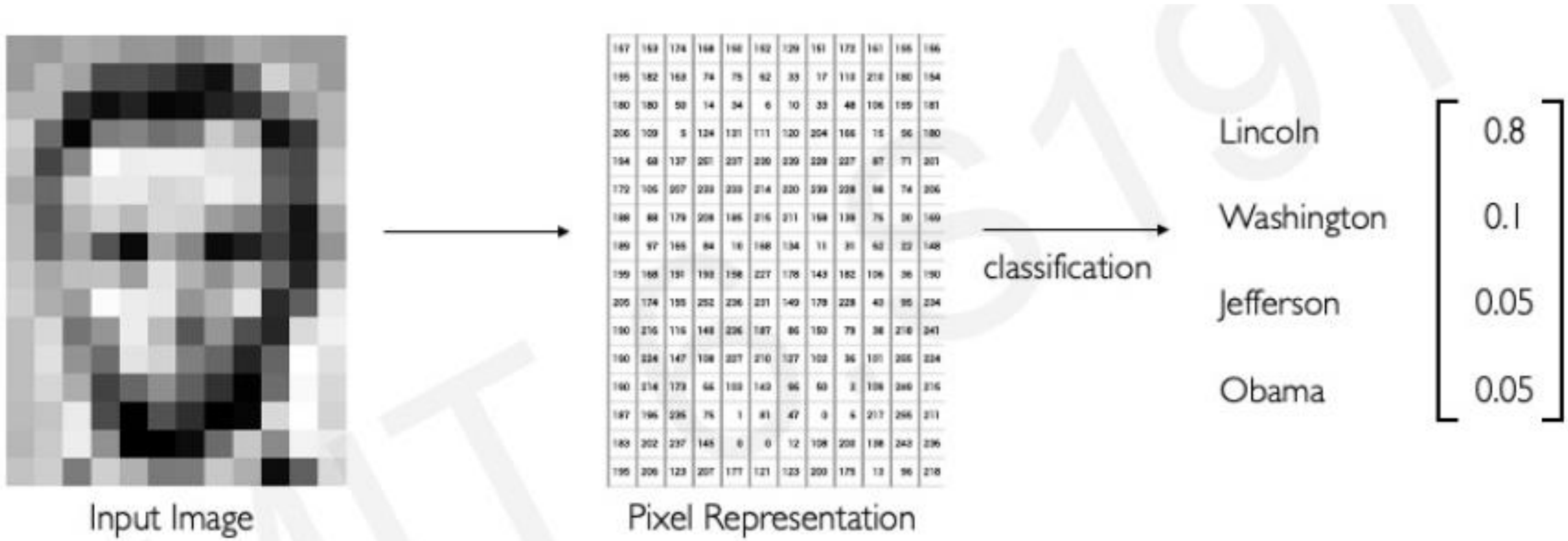


What the computer sees

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	106	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

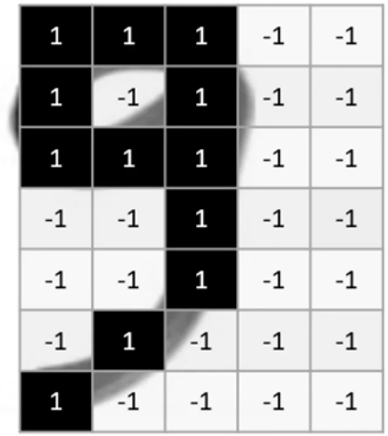
An image is just a matrix of numbers [0,255]!
i.e., 1080x1080x3 for an RGB image

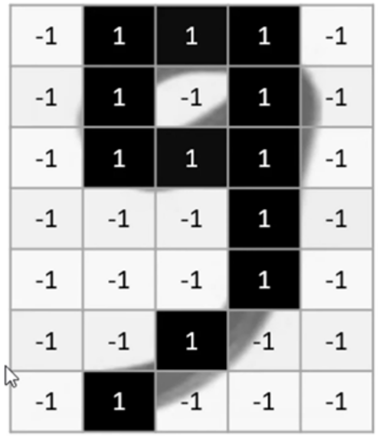
Image Classification Example



Variation of image representation

- Handwritten digit is represented as a grid of numbers (-1 and 1) or rgb numbers from 0 to 255
- The issue with this presentation is that is sensitive to a little shift in digit 9 (e.g., left vs. middle)
  - Computer will not be able to recognize that this is number 9 since any variation in the handwritten digit changes its two-dimensional representation





Learning from large image requires big network!

- Bigger image having 1920 by 1080 pixels would require six million neurons on the input layer (we need a rgb channel for red, green and blue)
- With a hidden layer of 4 million neurons you're talking about 24 million weights to be calculated just between the input and hidden layer!

Image size = 1920 x 1080 X 3

First layer neurons = 1920 x 1080 X 3 ~ 6 million

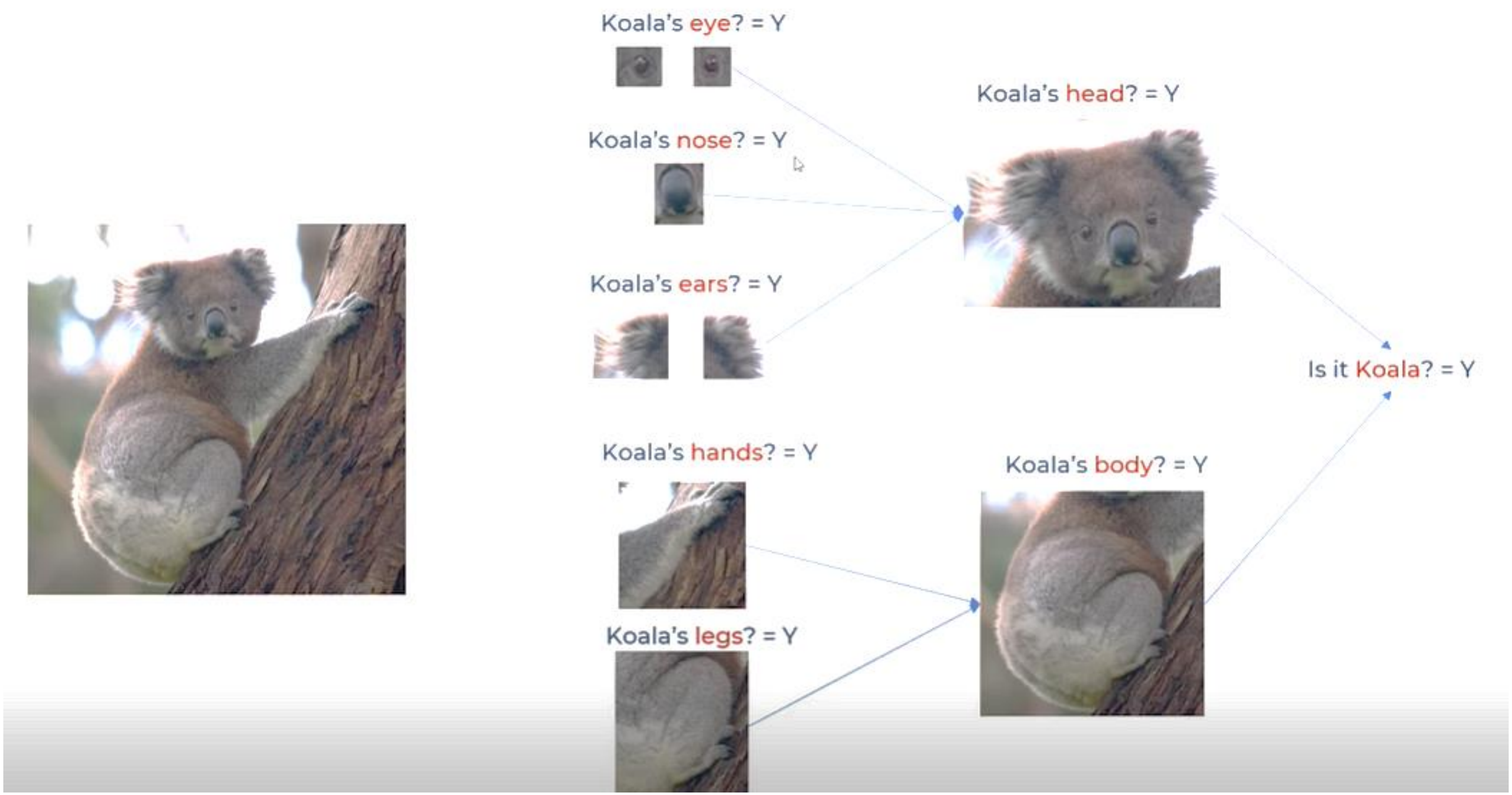
Hidden layer neurons = Let's say you keep it ~ 4 million

Weights between input and hidden layer = 6 mil * 4 mil
= 24 million

Limitations of FFN

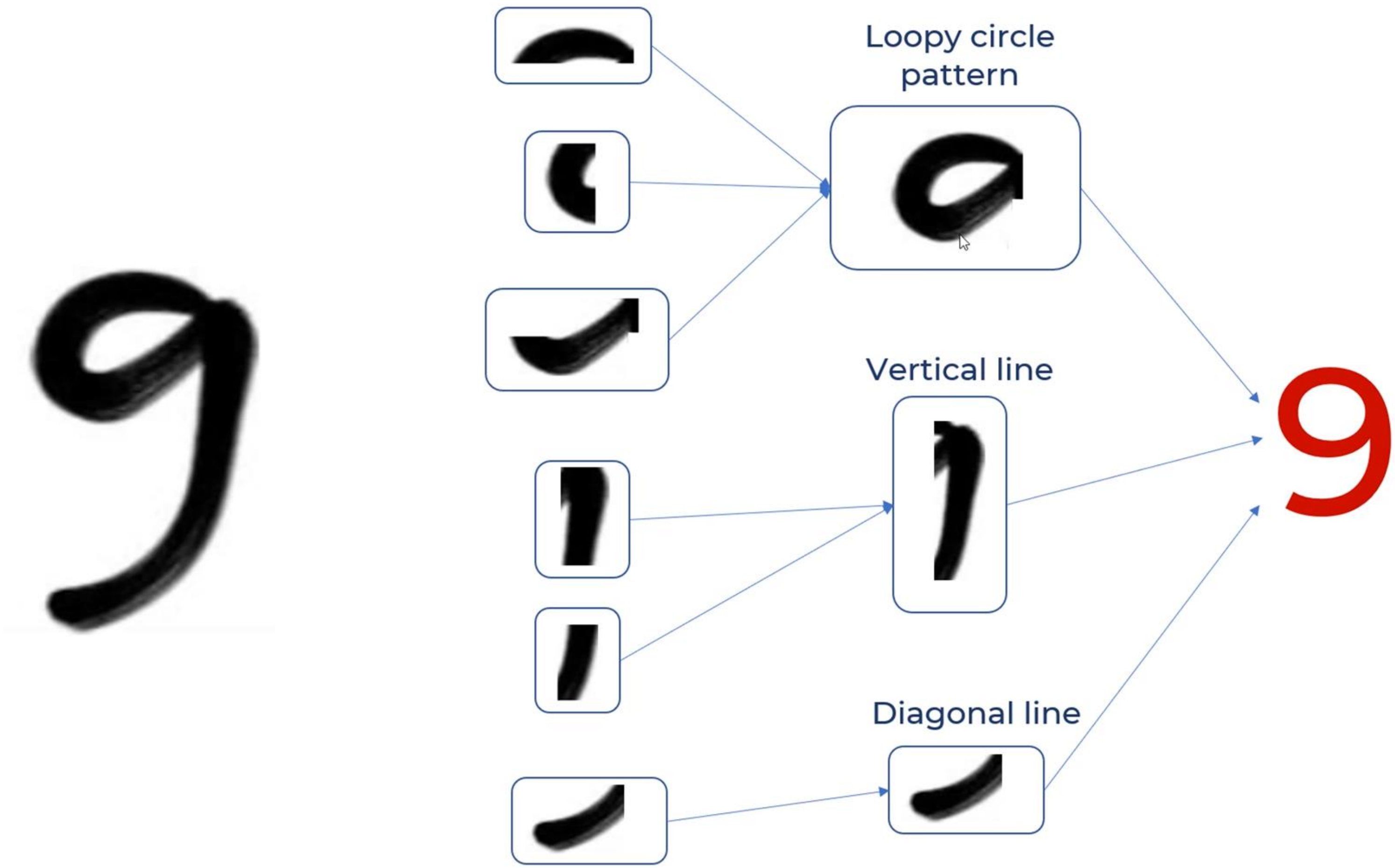
- Too much computation
- Treats local pixels same as pixels far apart
- Sensitive to location of an object in an image

Recognize Koala



- When we look at the Koala image, we look at the **small features** like round eyes, black nose, and fluffy ears and that triggered this as the Koala head by the different neurons in the brain and **<u>then aggregate the results and say the Koala head</u>**
- Similarly, we look up for the hands and legs and we say it is the body of the Koala body, and at the end by these features, our brains say it is the image of the Koala

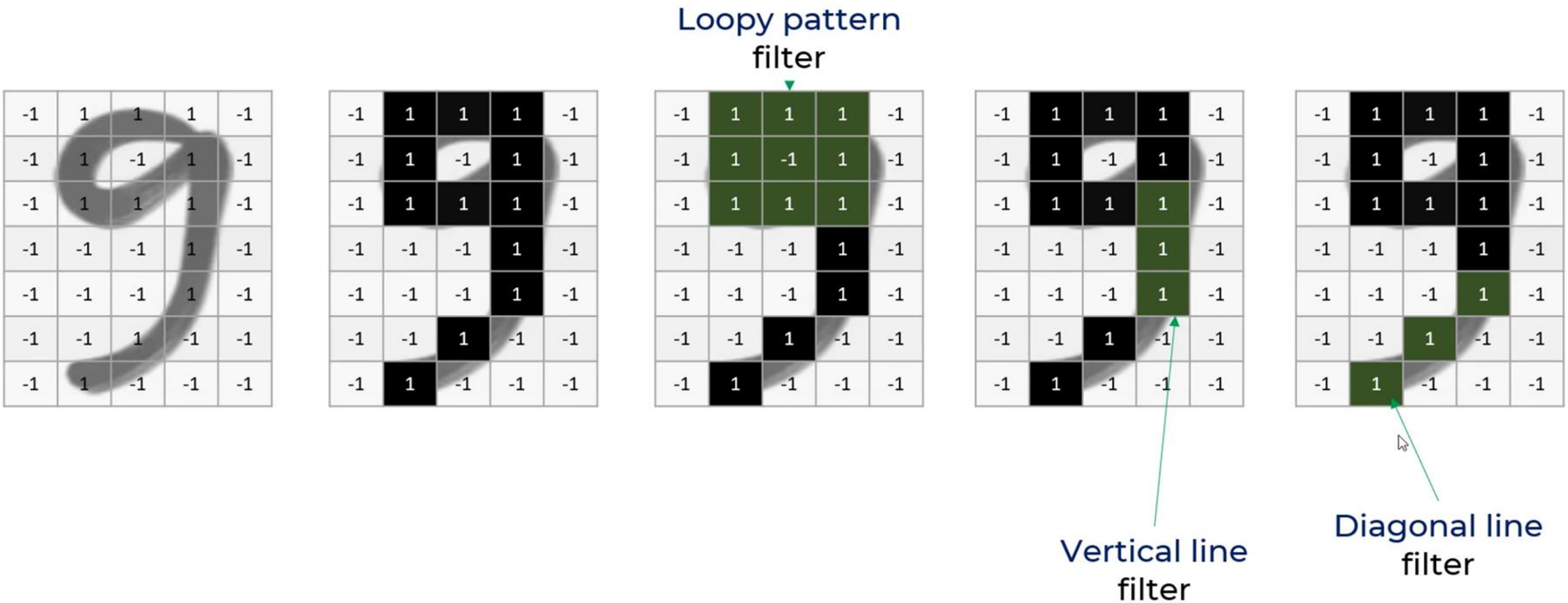
Recognize digits



To recognize 9, we need to:
- Detect edges that make-up the loopy circle pattern on top
- In the middle we detect a vertical line and at the bottom we detect a diagonal line

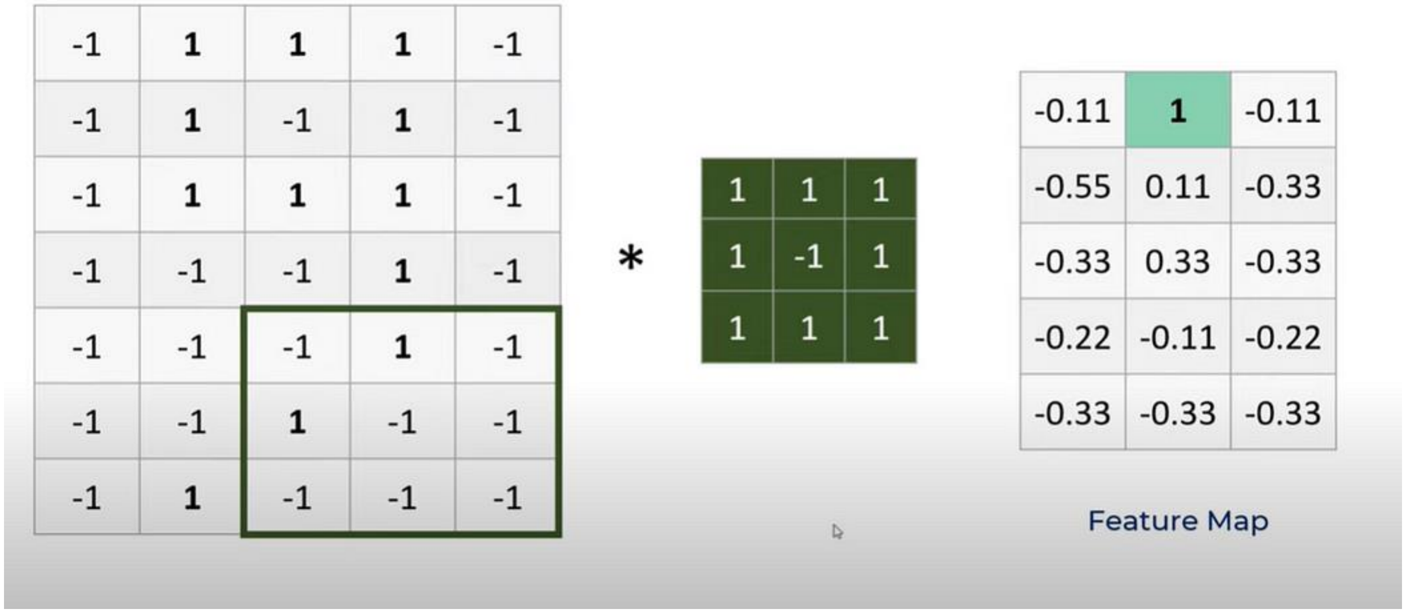
Detect Patterns

- For a NN to do so, we use the concept of **filters** which are the feature detectors



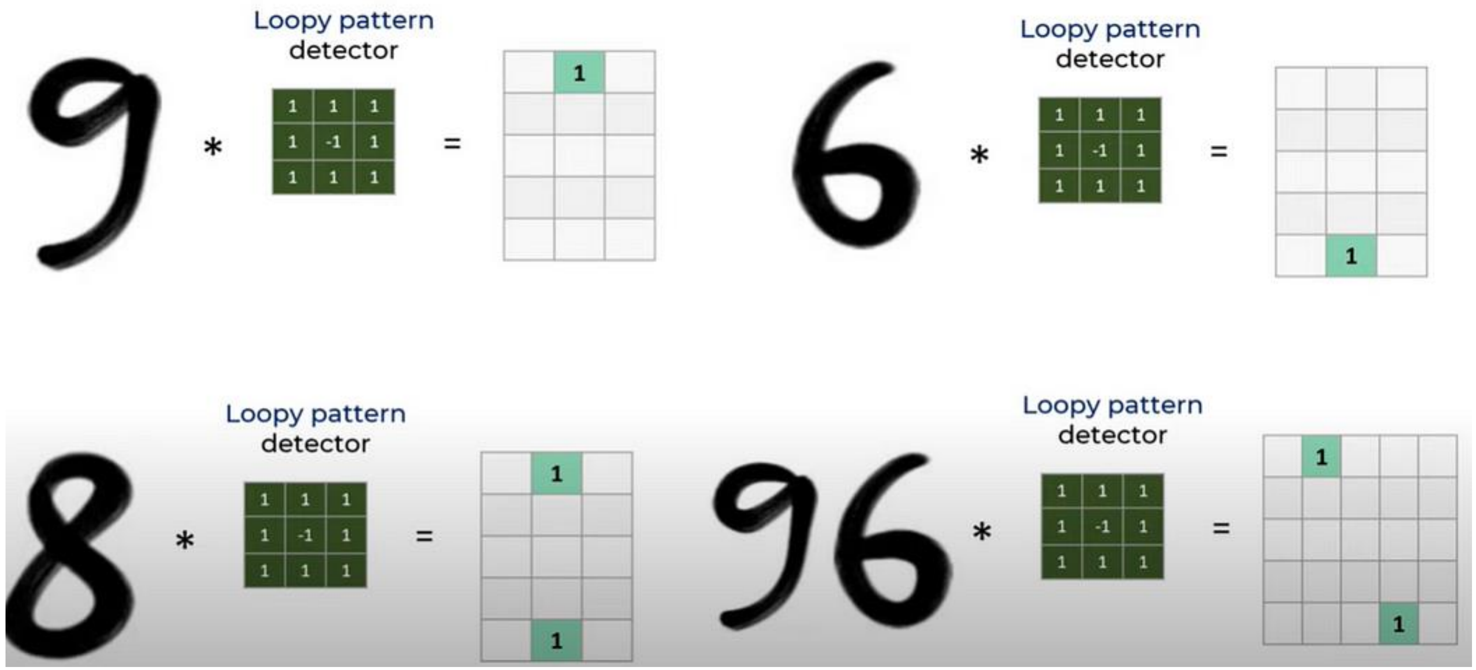
In the case of 9, we use **three filters**. Starting with head i.e., Loppy filter, then at the middle: the vertical line filter and at the bottom Diagonal line filter

Original image * Filter = Feature Map



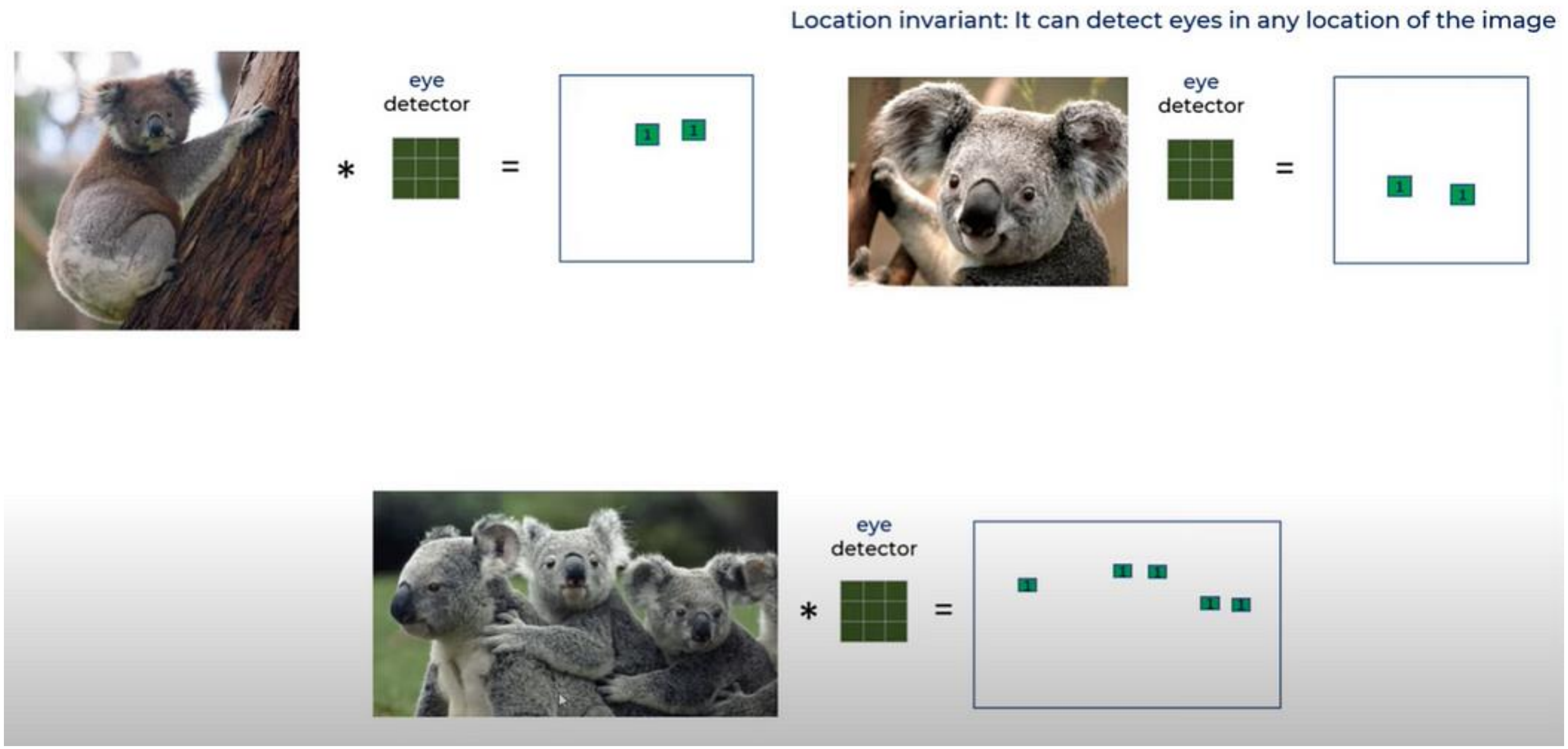
- On the original image which is 7x7 matrix and we perform a convolutional or **filter operation** with the loopy circle pattern in order to get feature map
- Multiply the subset of original image(3x3) with the filter and continue traversing the entire image and update the average in the feature map
- Allows position invariant feature detection

Role of filters



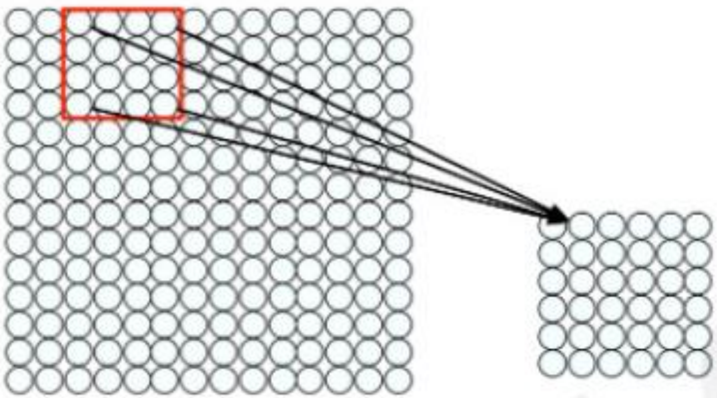
- The loopy circle is detected at the top for number 9, at the bottom for 6, both at the top and bottom in case of 8

Different position of eye



- Filter is moved all over the image to detect Eye at the different location => location are invariant as the
- In case of last image, it detected eye at the three different location, because their exist three koala bear

Feature Extraction with Convolution



- Filter of size 4x4 : 16 different weights
- Apply this same filter to 4x4 patches in input
- Shift by 2 pixels for next patch

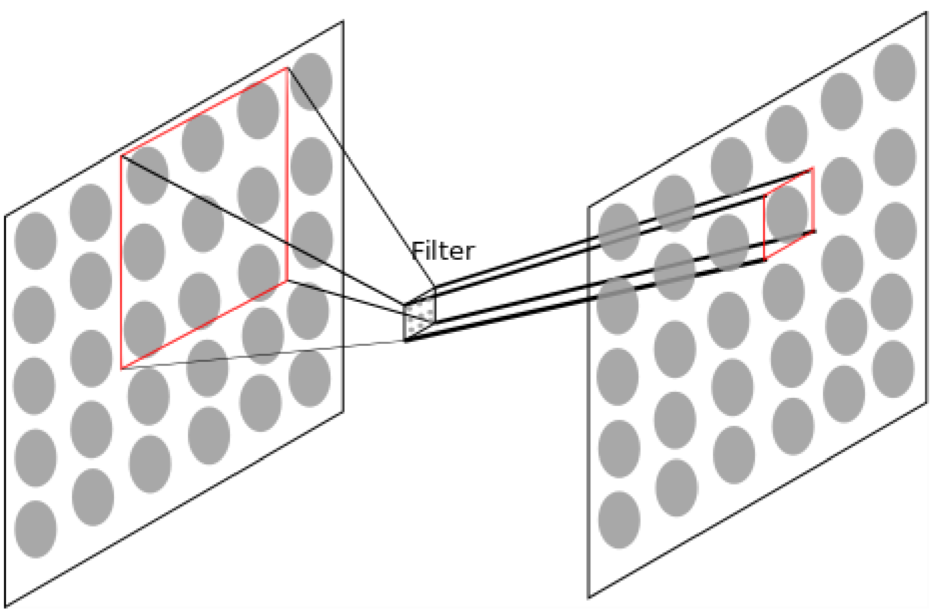
This "patchy" operation is **convolution**

1) Apply a set of weights – a filter – to extract **local features**

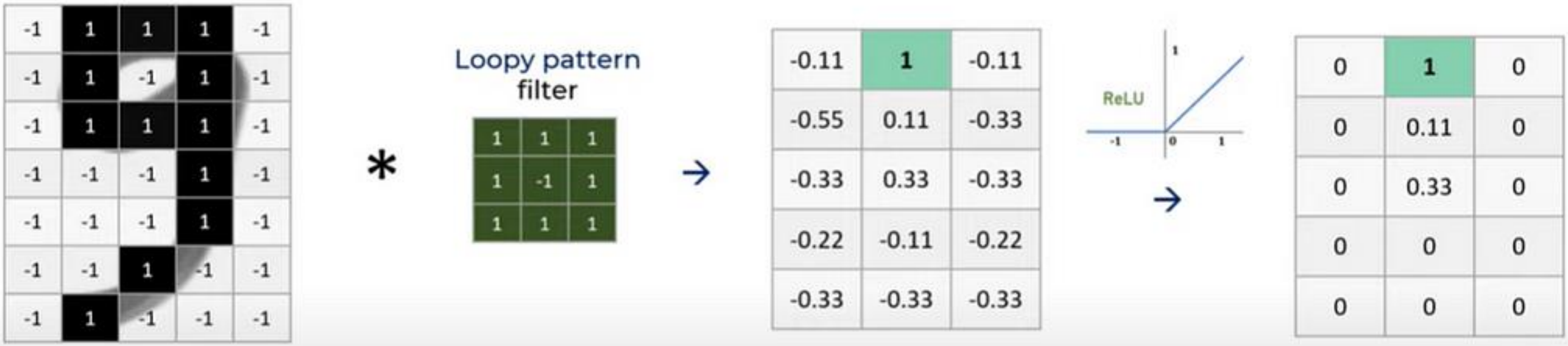
2) Use **multiple filters** to extract different features

Convolution of filters with input

- The network learns what weights to use from data
- The first layer filters learn edges
- Higher level filters learn more complex features

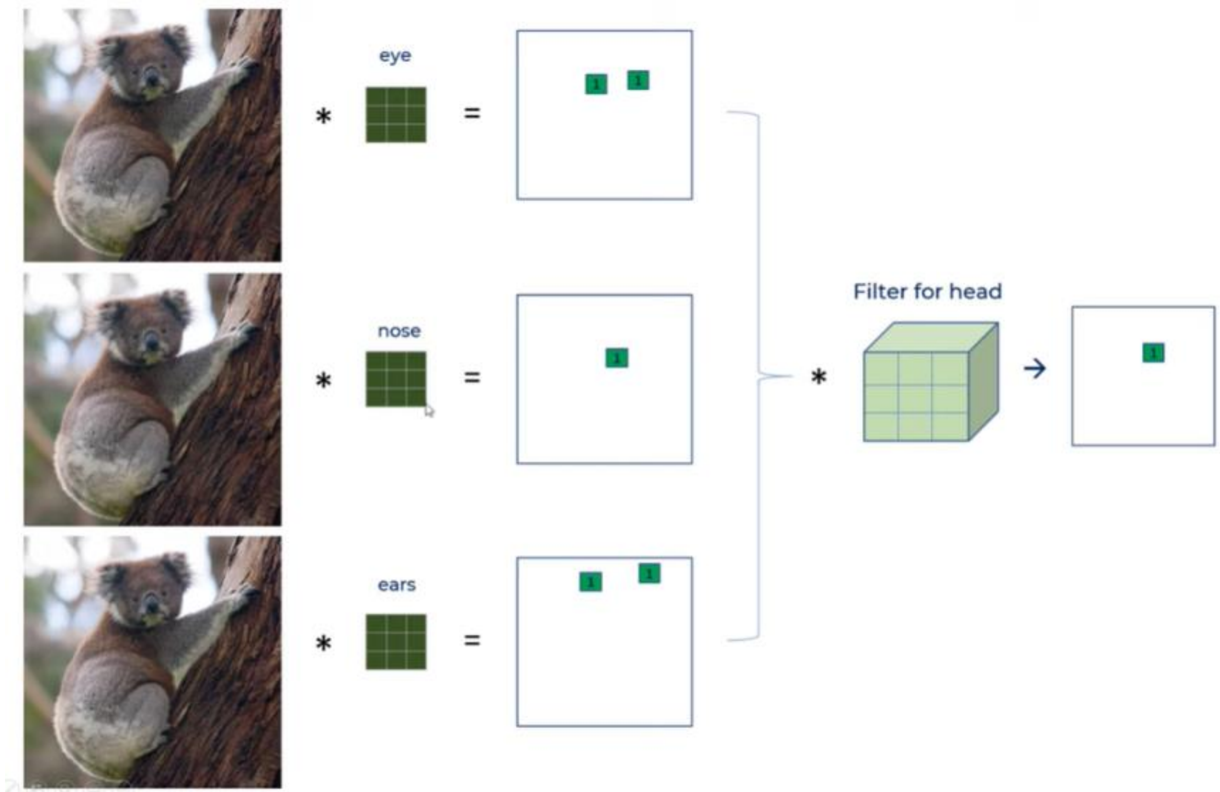


ReLU activation Function



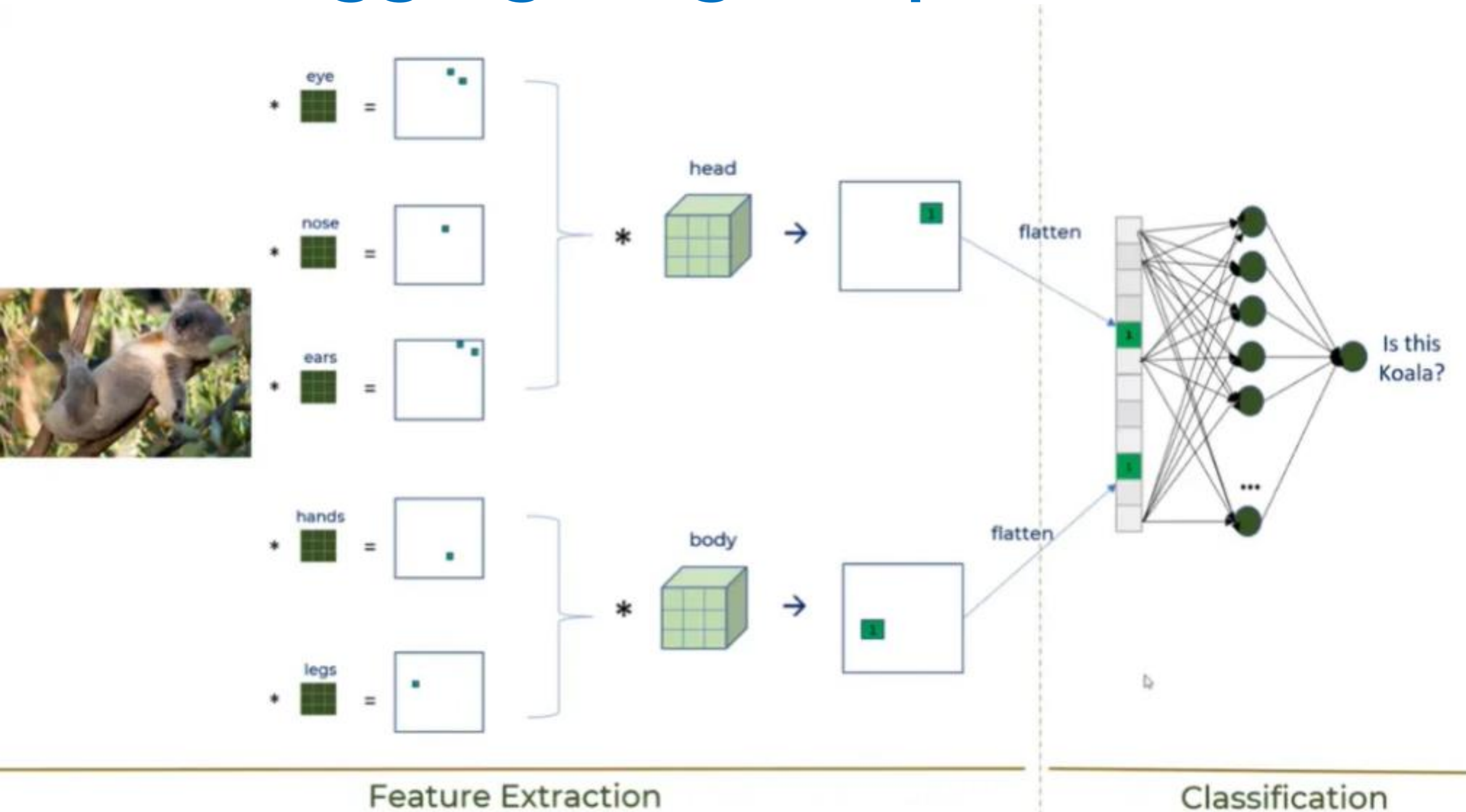
- Apply ReLu to introduce non-linearity in the model
    - It takes the feature map and changes the negative value to zero

Different filters for different features



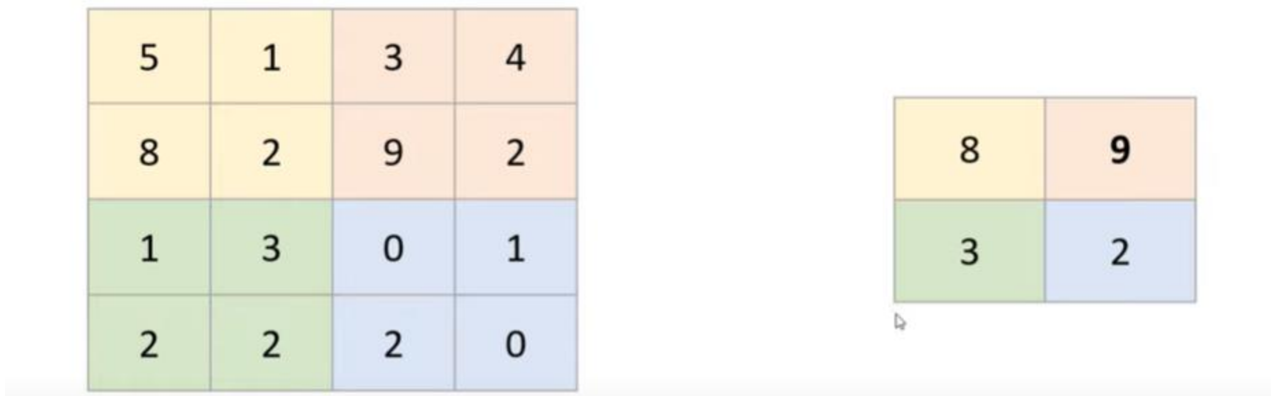
- Nose detector + nose detector + ear detector **=>** apply Convolution operation again to aggregate all these filters into head detector

Aggregating the patterns



- koala body detector + head detector => classify the image as Koala or not

Pooling Layer



- 2 by 2 pooling with stride 2
- Pooling layer is used to reduce the size of the image
- Two types of pooling can be done
  - Max Pooling
  - Average Pooling

Conv and Pooling animations

- CNN Conv

  https://deeplizard.com/resource/pavq7noze2

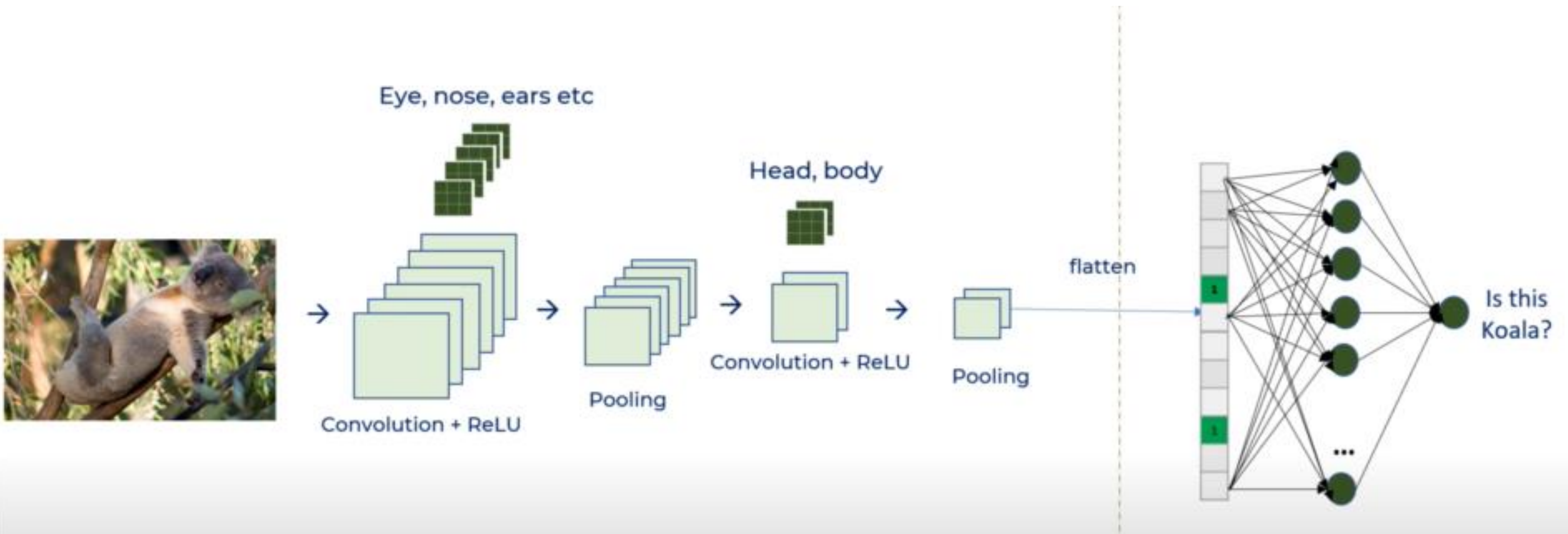
- CNN MaxPool

  https://deeplizard.com/resource/pavq7noze3

CNN

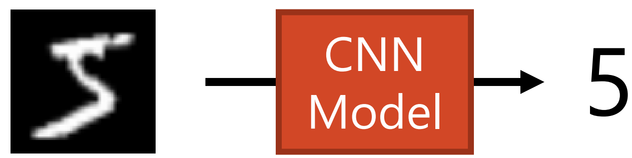
- Putting it all together:

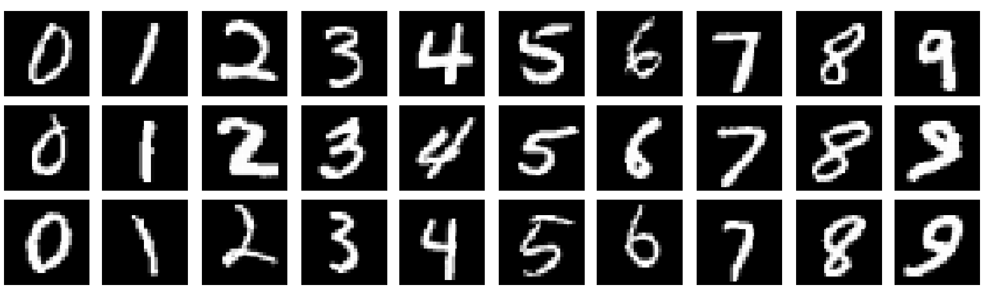
  {Convolution, ReLU, Pooling}N, Fully-connected, Softmax
- Overall Architecture of Koala Feature extraction and Classification



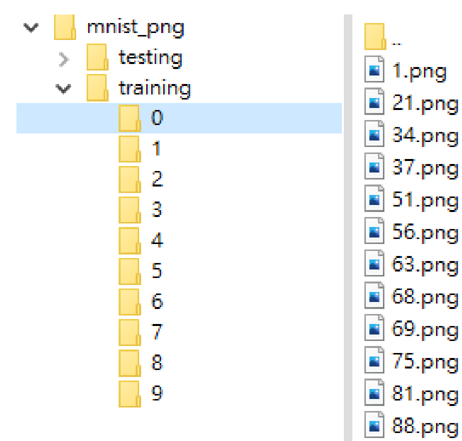
Handwritten Digit Classification

- Objective
  - train a CNN which can recognize a hand-written digit





Dataset directory



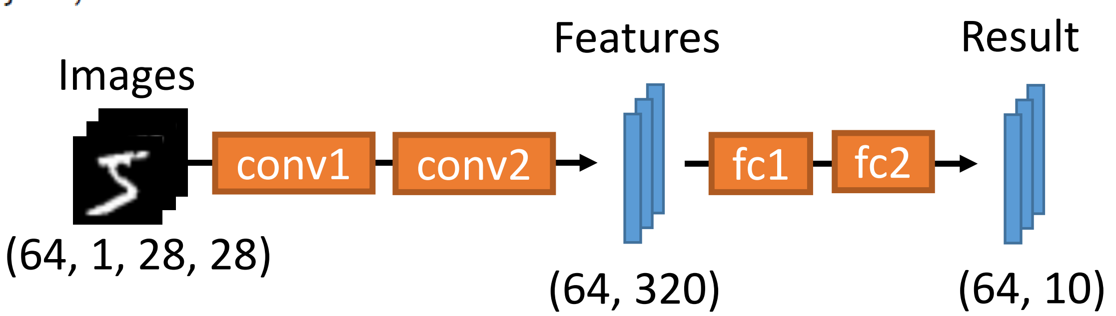
Create a CNN model named “Net” inherited from Pytorch in-build nn.Module class

1. Define Model

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 10, kernel_size=5),
            nn.MaxPool2d(2),
            nn.ReLU()
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(10, 20, kernel_size=5),
            nn.Dropout2d(0.5),
            nn.MaxPool2d(2),
            nn.ReLU()
        )
        self.fc1 = nn.Sequential(
            nn.Linear(320, 50),
            nn.ReLU(),
            nn.Dropout(0.5)
        )
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(-1, 320)
        x = self.fc1(x)
        x = self.fc2(x)
        return x
```

Self-define a layer with various built-in layers



2. Train Model

```
def train(model, epoch, log_interval=100):
    optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
    criterion = nn.CrossEntropyLoss()
    model.train()  # Important: set training mode

    iteration = 0
    for ep in range(epoch):
        for batch_idx, (data, target) in enumerate(trainset_loader):
            data, target = data.cuda(), target.cuda()
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

            if iteration % log_interval == 0:
                print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                    ep, batch_idx * len(data), len(trainset_loader.dataset),
                    100. * batch_idx / len(trainset_loader), loss.item()))
            iteration += 1
```

In every epoch, the whole dataset would be split into several batches and the network would be trained batch-by-batch.

Forward propagation and calculate the loss

Backward propagation (**autograd**)

Update the network

```
Train Epoch: 0 [0/60000 (0%)]	Loss: 2.315348
Train Epoch: 0 [6400/60000 (11%)]	Loss: 2.307844
Train Epoch: 0 [12800/60000 (21%)]	Loss: 2.281325
Train Epoch: 0 [19200/60000 (32%)]	Loss: 2.291601
Train Epoch: 0 [25600/60000 (43%)]	Loss: 2.262873
Train Epoch: 0 [32000/60000 (53%)]	Loss: 2.211873
Train Epoch: 0 [38400/60000 (64%)]	Loss: 2.134070
Train Epoch: 0 [44800/60000 (75%)]	Loss: 1.865652
Train Epoch: 0 [51200/60000 (85%)]	Loss: 1.624762
Train Epoch: 0 [57600/60000 (96%)]	Loss: 1.447150

Test set: Average loss: 0.0009, Accuracy: 7732/10000 (77%)
```

```
Train Epoch: 4 [3072/60000 (5%)]	Loss: 0.181250
Train Epoch: 4 [9472/60000 (16%)]	Loss: 0.192755
Train Epoch: 4 [15872/60000 (26%)]	Loss: 0.255178
Train Epoch: 4 [22272/60000 (37%)]	Loss: 0.073243
Train Epoch: 4 [28672/60000 (48%)]	Loss: 0.169505
Train Epoch: 4 [35072/60000 (58%)]	Loss: 0.188644
Train Epoch: 4 [41472/60000 (69%)]	Loss: 0.029532
Train Epoch: 4 [47872/60000 (80%)]	Loss: 0.127380
Train Epoch: 4 [54272/60000 (90%)]	Loss: 0.193711

Test set: Average loss: 0.0001, Accuracy: 9680/10000 (97%)
```

3. Validate Model

```
def test(model):
    criterion = nn.CrossEntropyLoss()
    model.eval()  # Important: set evaluation mode
    test_loss = 0
    correct = 0

    for data, target in testset_loader:
        data, target = data.cuda(), target.cuda()
        output = model(data)
        test_loss += criterion(output, target).item() # sum up batch loss
        pred = output.max(1, keepdim=True)[1] # get the index of the max log-probability
        correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(testset_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(testset_loader.dataset),
        100. * correct / len(testset_loader.dataset)))
```

We use both accuracy and xe-loss to evaluate the performance of network for the testing data.

Summary

- **Convolution**
  - Can we learn a hierarchy of features directly from the data instead of hand engineering
  - Enables location invariant feature detection
- **ReLu**
  - Introduces nonlinearity
  - Speeds up training, faster to compute
- **Pooling**
  - Reduces dimensions and computation
  - Reduces overfitting
  - Makes the model tolerant towards small distortion and variations