Deep Learning

Autonomous Robotics Lab

Contact: salanvoj@fel.cvut.cz

Motivation

Detect and localize barbie in unknown environment

Timeline

- Where we will do it
    - servers
- How we will do it
    - Pytorch
- What we will do
    - detector

Before we start

- There are two GPU servers for students

  cantor.felk.cvut.cz

  taylor.felk.cvut.cz

- You can access them using ssh command

- Information is on website:

cw.fel.cvut.cz/b202/courses/aro/tutorials/learning_on_gpu_servers

GPU servers

- Important command: “nvidia-smi” – shows actual load on GPUs

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 410.48                 Driver Version: 410.48                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 108...  On   | 00000000:04:00.0 Off |                  N/A |
| 43%   64C    P2   310W / 250W |  10366MiB / 11178MiB |     51%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 108...  On   | 00000000:05:00.0 Off |                  N/A |
| 52%   68C    P2   208W / 250W |   8227MiB / 11178MiB |     40%      Default |
+-------------------------------+----------------------+----------------------+
|   2  GeForce GTX 108...  On   | 00000000:08:00.0 Off |                  N/A |
| 29%   37C    P8    15W / 250W |      0MiB / 11178MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  GeForce GTX 108...  On   | 00000000:09:00.0 Off |                  N/A |
| 29%   31C    P8    15W / 250W |      0MiB / 11178MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   4  GeForce GTX 108...  On   | 00000000:84:00.0 Off |                  N/A |
| 29%   31C    P8    15W / 250W |      0MiB / 11178MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   5  GeForce GTX 108...  On   | 00000000:85:00.0 Off |                  N/A |
| 29%   35C    P8    15W / 250W |      0MiB / 11178MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   6  GeForce GTX 108...  On   | 00000000:88:00.0 Off |                  N/A |
| 29%   32C    P8    14W / 250W |      0MiB / 11178MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

GPU servers

- Always choose the card with enough memory with command:

  export CUDA_VISIBLE_DEVICES=X

  where X is the number of selected card

Working environment on GPU servers

- You have to activate virtual environment to use pytorch on GPU servers

```
source /opt/torchenv/bin/activate
```

PyTorch

https://pytorch.org/tutorials/

Tensors

Tensors are similar to NumPy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

```
import torch
x = torch.tensor([5.5, 3]
print(x)
```

PyTorch

From numpy to tensor

```
nparr = np.array([5.5, 3])
x = torch.from_numpy(nparr)
```

From tensor to numpy

```
nparr = x.numpy()
```

PyTorch

Computation graph

a = torch.tensor(1)

b = torch.tensor(3.)

c = a + b or c = torch.add(a, b)

c = a * b or c = torch.mul(a, b)

PyTorch

Tensors has attribute .requires_grad

-> if it is set to True pytorch tracks all operations on this tensor

x = torch.rand(2, 2, requires_grad=True)

-> then we can use backward() function to compute gradients

Training

```
import numpy as np                                                          # Imports
import torch

X = np.random.rand(30, 1)*2.0                                               # Create data as numpy arrays
w = np.random.rand(2, 1)                                                    # Randomly select weights
y = (X*w[0] + w[1]/(X+1)**2) + np.random.randn(30, 1) * 0.05                # Create targets
print('target w1 {} w2 {}'.format(w[0], w[1]))
Xt = torch.from_numpy(X).float()                                            # Convert numpy arrays to torch tensors
yt = torch.from_numpy(y).float()
W1 = torch.rand(1, requires_grad=True)                                      # Initialize weights randomly
W2 = torch.rand(1, requires_grad=True)

lr = 0.005                                                                  # set up learning rate
for epoch in range(2500):
    y_pred = (torch.mul(W1,Xt), torch.div(W2, torch.add(Xt,torch.tensor(1))**2))   # Compute predictions
    loss = torch.mean((y_pred - yt) ** 2)                                   # Compute cost function
    loss.backward()                                                         # Run back-propagation
    W1.data = W1.data - lr*W1.grad.data                                     # Update parameters
    W2.data = W2.data - lr*W2.grad.data
    W1.grad.data.zero_()                                                    # Reset gradients
    W2.grad.data.zero_()

print('found w {} b {}'.format(W1.data ,W2.data))
```

Training

Model with parameters

```
import numpy as np                                                          # Imports
import torch

X = np.random.rand(30, 1)*2.0                                               # Create data as numpy arrays
w = np.random.rand(2, 1)                                                    # Randomly select weights
y = (X*w[0] + w[1]/(X+1)**2) + np.random.randn(30, 1) * 0.05                # Create targets
print('target w1 {} w2 {}'.format(w[0], w[1]))
Xt = torch.from_numpy(X).float()                                            # Convert numpy arrays to torch tensors
yt = torch.from_numpy(y).float()
W1 = torch.rand(1, requires_grad=True)                                      # Initialize weights randomly
W2 = torch.rand(1, requires_grad=True)

lr = 0.005                                                                  # set up learning rate
for epoch in range(2500):
    y_pred = (torch.mul(W1,Xt), torch.div(W2, torch.add(Xt,torch.tensor(1))**2))   # Compute predictions
    loss = torch.mean((y_pred - yt) ** 2)                                   # Compute cost function
    loss.backward()                                                         # Run back-propagation
    W1.data = W1.data - lr*W1.grad.data                                     # Update parameters
    W2.data = W2.data - lr*W2.grad.data
    W1.grad.data.zero_()                                                    # Reset gradients
    W2.grad.data.zero_()

print('found w {} b {}'.format(W1.data ,W2.data))
```

Training

Loss computation

```
import numpy as np                                                                  # Imports
import torch

X = np.random.rand(30, 1)*2.0                                                       # Create data as numpy arrays
w = np.random.rand(2, 1)                                                            # Randomly select weights
y = (X*w[0] + w[1]/(X+1)**2) + np.random.randn(30, 1) * 0.05                        # Create targets
print('target w1 {} w2 {}'.format(w[0], w[1]))
Xt = torch.from_numpy(X).float()                                                    # Convert numpy arrays to torch tensors
yt = torch.from_numpy(y).float()
W1 = torch.rand(1, requires_grad=True)                                              # Initialize weights randomly
W2 = torch.rand(1, requires_grad=True)

lr = 0.005                                                                          # set up learning rate
for epoch in range(2500):
    y_pred = (torch.mul(W1,Xt), torch.div(W2, torch.add(Xt,torch.tensor(1))**2))    # Compute predictions
    loss = torch.mean((y_pred - yt) ** 2)                                           # Compute cost function
    loss.backward()                                                                 # Run back-propagation
    W1.data = W1.data - lr*W1.grad.data                                             # Update parameters
    W2.data = W2.data - lr*W2.grad.data
    W1.grad.data.zero_()                                                            # Reset gradients
    W2.grad.data.zero_()

print('found w {} b {}'.format(W1.data ,W2.data))
```

Training

Backward pass (gradient computation)

```
import numpy as np                                                          # Imports
import torch

X = np.random.rand(30, 1)*2.0                                               # Create data as numpy arrays
w = np.random.rand(2, 1)                                                    # Randomly select weights
y = (X*w[0] + w[1]/(X+1)**2) + np.random.randn(30, 1) * 0.05                # Create targets
print('target w1 {} w2 {}'.format(w[0], w[1]))
Xt = torch.from_numpy(X).float()                                            # Convert numpy arrays to torch tensors
yt = torch.from_numpy(y).float()
W1 = torch.rand(1, requires_grad=True)                                      # Initialize weights randomly
W2 = torch.rand(1, requires_grad=True)

lr = 0.005                                                                  # set up learning rate
for epoch in range(2500):
    y_pred = (torch.mul(W1,Xt), torch.div(W2, torch.add(Xt,torch.tensor(1))**2))   # Compute predictions
    loss = torch.mean((y_pred - yt) ** 2)                                   # Compute cost function
    loss.backward()                                                         # Run back-propagation
    W1.data = W1.data - lr*W1.grad.data                                     # Update parameters
    W2.data = W2.data - lr*W2.grad.data
    W1.grad.data.zero_()                                                    # Reset gradients
    W2.grad.data.zero_()

print('found w {} b {}'.format(W1.data ,W2.data))
```

Training

Optimization step

```
import numpy as np                                                          # Imports
import torch

X = np.random.rand(30, 1)*2.0                                               # Create data as numpy arrays
w = np.random.rand(2, 1)                                                    # Randomly select weights
y = (X*w[0] + w[1]/(X+1)**2) + np.random.randn(30, 1) * 0.05                # Create targets
print('target w1 {} w2 {}'.format(w[0], w[1]))
Xt = torch.from_numpy(X).float()                                            # Convert numpy arrays to torch tensors
yt = torch.from_numpy(y).float()
W1 = torch.rand(1, requires_grad=True)                                      # Initialize weights randomly
W2 = torch.rand(1, requires_grad=True)

lr = 0.005                                                                  # set up learning rate
for epoch in range(2500):
    y_pred = (torch.mul(W1,Xt), torch.div(W2, torch.add(Xt,torch.tensor(1))**2))   # Compute predictions
    loss = torch.mean((y_pred - yt) ** 2)                                   # Compute cost function
    loss.backward()                                                         # Run back-propagation
    W1.data = W1.data - lr*W1.grad.data                                     # Update parameters
    W2.data = W2.data - lr*W2.grad.data
    W1.grad.data.zero_()                                                    # Reset gradients
    W2.grad.data.zero_()

print('found w {} b {}'.format(W1.data ,W2.data))
```

Training

Optimization step

This could be replaced by optimizer

```
optimizer = torch.optim.SGD(parameters, lr, momentum, weight_decay)
optimizer.step()                                                    # Update parameters
optimizer.zero_grad()                                               # Reset gradients
```

```
W1.data = W1.data - lr*W1.grad.data                                 # Update parameters
W2.data = W2.data - lr*W2.grad.data
W1.grad.data.zero_()                                                # Reset gradients
W2.grad.data.zero_()
```

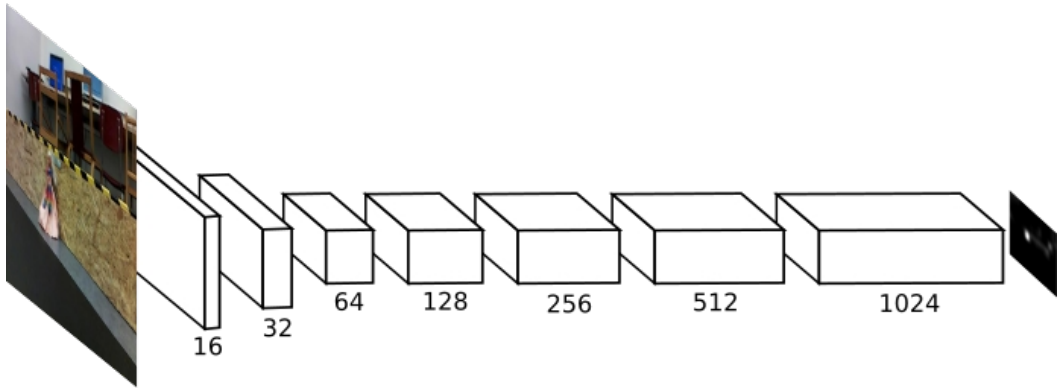
Barbie detector

- Our goal is to detect barbie in image and estimate a 3d coordinates of that barbie.

X, Y, Z

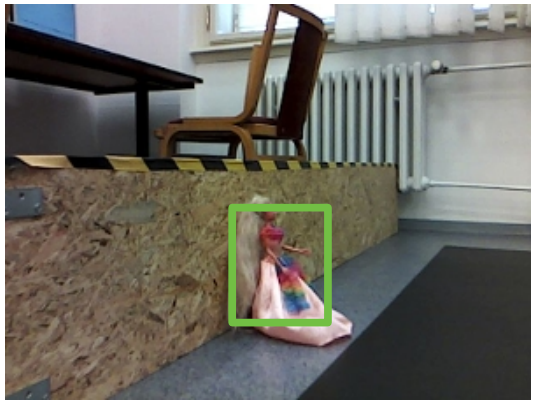
Model

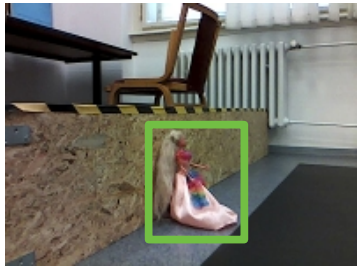
- We use architecture of the network based on YOLO V2 TINY



- We run the network on the multiple input image scales to detect barbies of all sizes

Network will find barbies size are near to 64x48 pixels

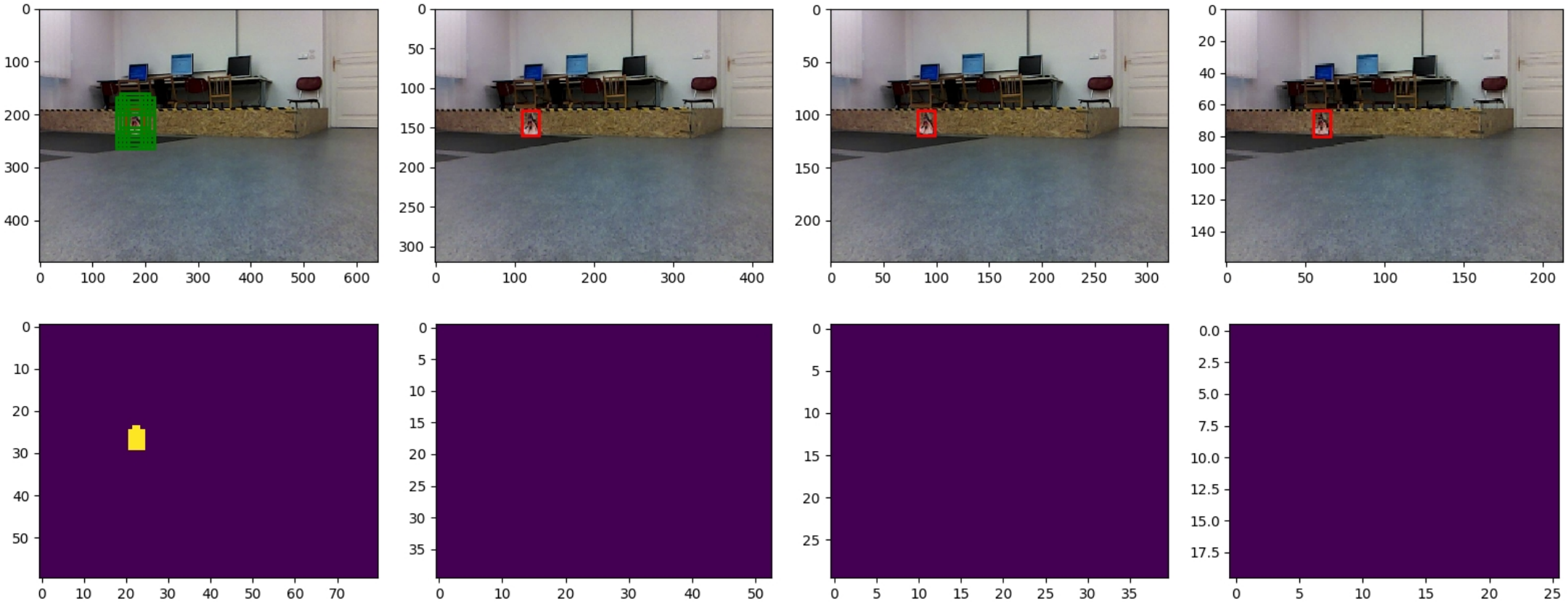




Training dataset

Input

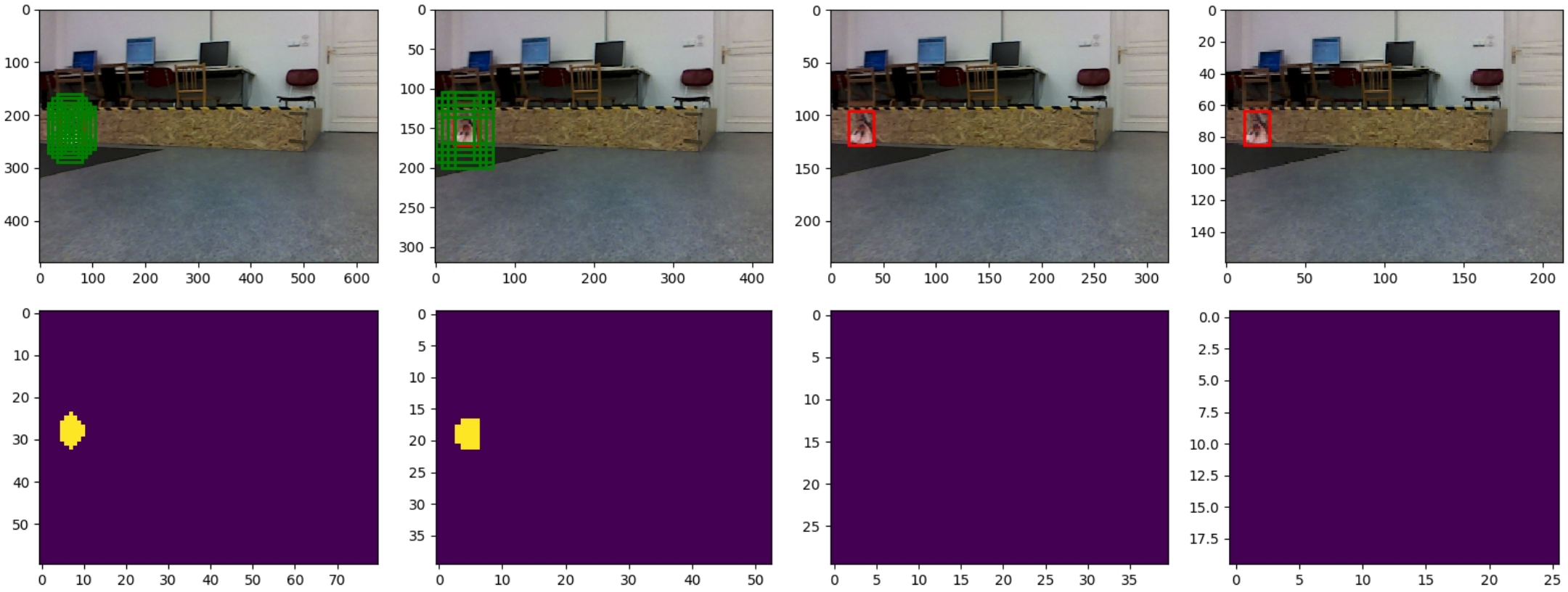
Label



Training dataset

Input

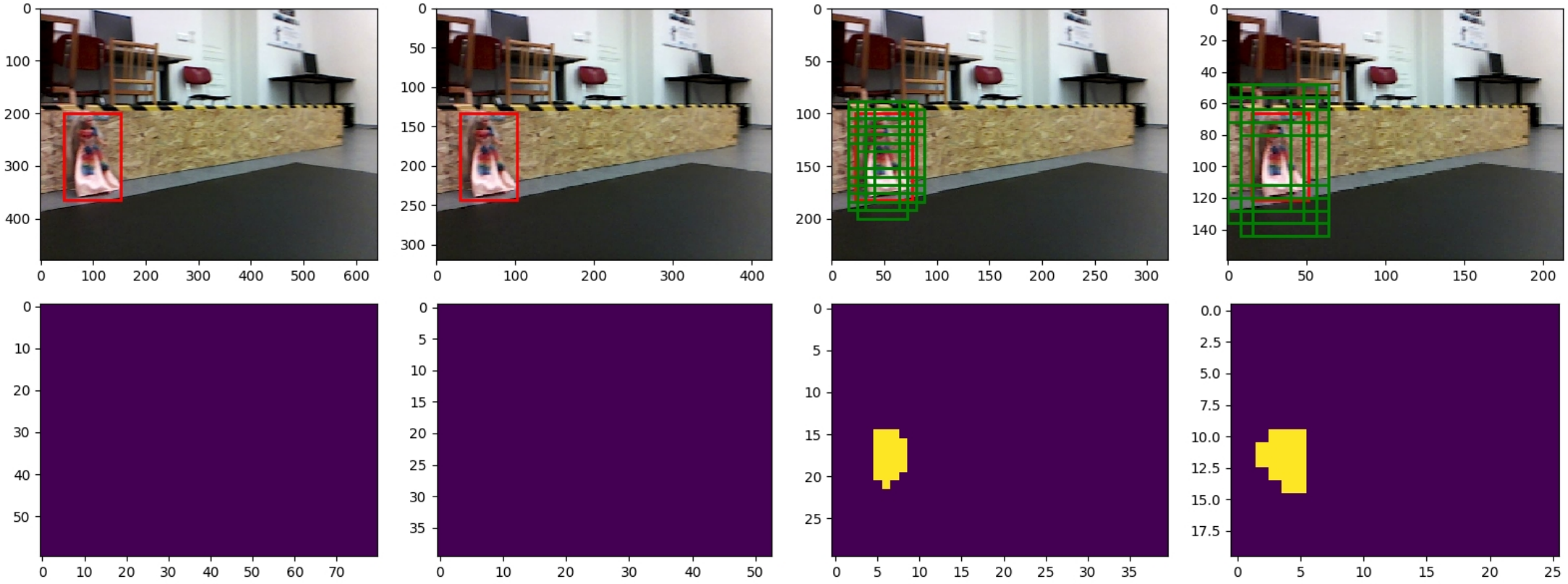
Label



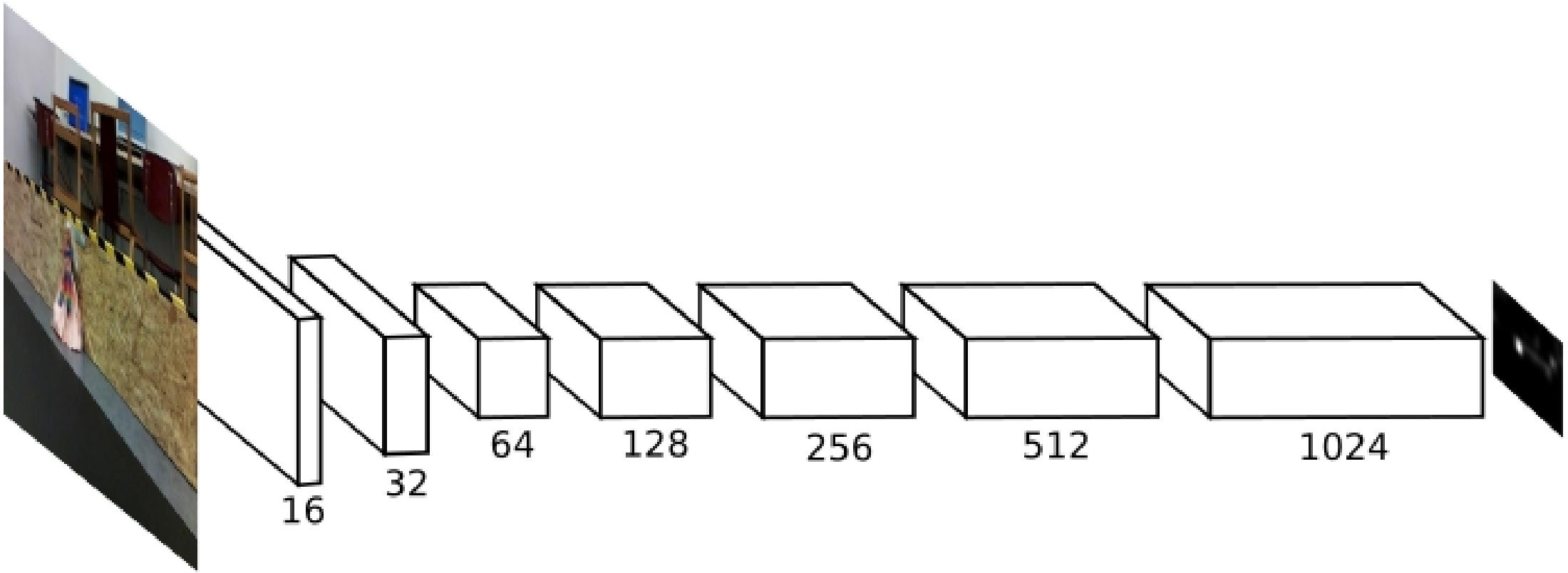
Training dataset

Input

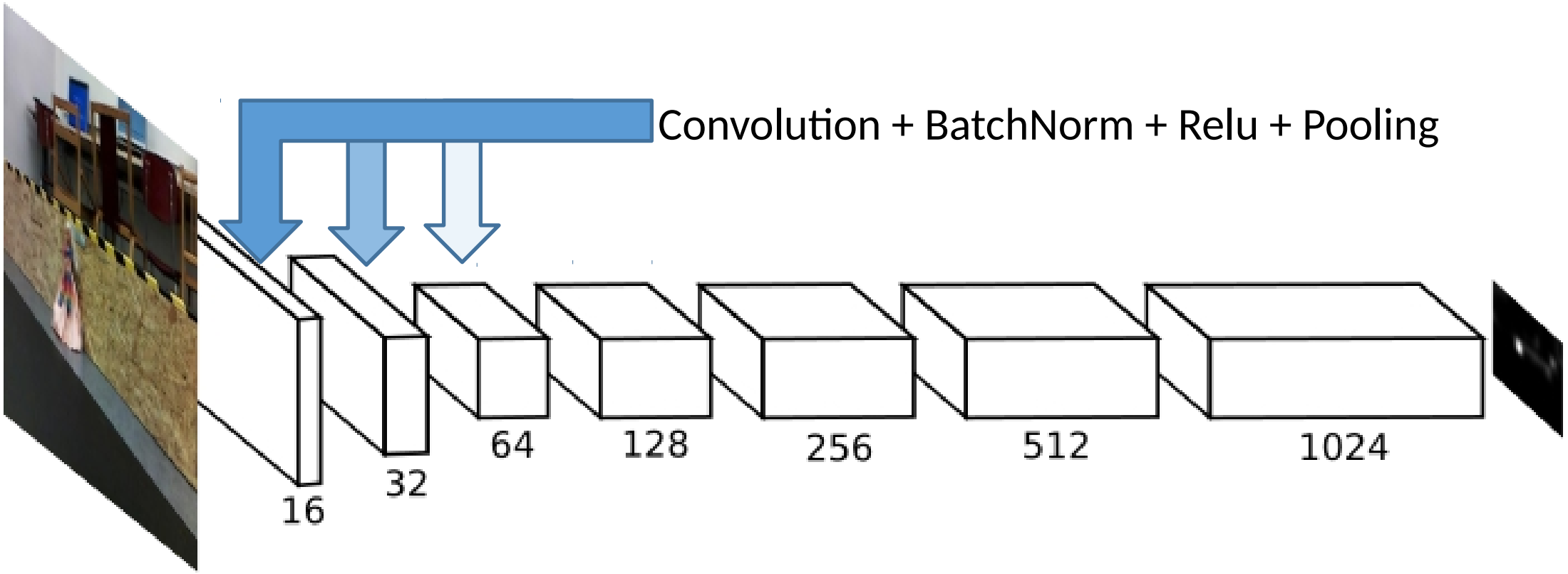
Label



Network architecture



Network architecture



Network class

- Class Net(nn.module) has two main functions

  __init__()

  defines neural network blocks and their parameters

  forward

  defines computation graph of neural network

def __init__(self):

super(Net, self).__init__()
self.relu = leaky relu layer
self.pool1 = max pooling layer with kernel 2x2 and stride 2
self.pool2 = max pooling layer with kernel 3x3, stride 1 and padding 1
self.bn16 = batch normalization layer on 16 channels
self.bn32 = batch normalization layer on 32 channels
self.bn64 = batch normalization layer on 64 channels
self.bn128 = batch normalization layer on 128 channels
self.bn256 = batch normalization layer on 256 channels
self.bn512 = batch normalization layer on 512 channels
self.bn1024 = batch normalization layer on 1024 channels
self.conv1 = convolutional layer without bias with 3 input_channels, 16 output_channels, 3x3 kernel, stride 1 and padding 1
self.conv2 = convolutional layer without bias with 16 input_channels, 32 output_channels, 3x3 kernel, stride 1 and padding 1
self.conv3 = convolutional layer without bias with 32 input_channels, 64 output_channels, 3x3 kernel, stride 1 and padding 1
self.conv4 = convolutional layer without bias with 64 input_channels, 128 output_channels, 3x3 kernel, stride 1 and padding 1
self.conv5 = convolutional layer without bias with 128 input_channels, 256 output_channels, 3x3 kernel, stride 1 and padding 1
self.conv6 = convolutional layer without bias with 256 input_channels, 512 output_channels, 3x3 kernel, stride 1 and padding 1
self.conv7 = convolutional layer without bias with 512 input_channels, 1024 output_channels, 3x3 kernel, stride 1 and padding 1
self.conv8 = convolutional layer with 1024 input_channels, 1 output_channel, 1x1 kernel, stride 1

def forward(self, input):

```
x = self.pool1(self.relu(self.bn16(self.conv1(input))))
x = self.pool1(self.relu(self.bn32(self.conv2(x))))
x = self.pool1(self.relu(self.bn64(self.conv3(x))))
x = self.pool2(self.relu(self.bn128(self.conv4(x))))
x = self.pool2(self.relu(self.bn256(self.conv5(x))))
x = self.pool2(self.relu(self.bn512(self.conv6(x))))
x = self.relu(self.bn1024(self.conv7(x)))
x = (self.conv8(x))
return x
```

Training script

- Training script is prepared for you!
  - You need to set paths for training and validation data
  - They are shared on the GPU servers in
    - /opt/barbie/barbie_validation_data
    - /opt/barbie/barbie_training_data

and set training parameters properly:

- batch_size
- learning_rate
- number of epochs
- flip_images

You can initialize network using pretrained weights and freeze the pretrained layers for number of epochs defined in variable:

- freeze_pretrained_layers

Training script

- When you lost a connection the running script will be terminated

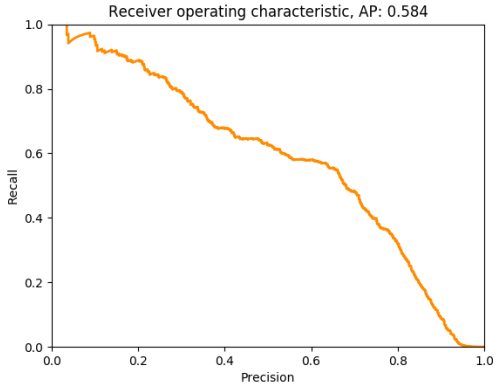
  -> use ”screen” when running the training script

Basic Linux Screen Usage

1. On the command prompt, type **screen** .
2. Run the desired program.
3. Use the key sequence Ctrl-a + Ctrl-d to detach from the **screen** session.
4. Reattach to the **screen** session by typing **screen** -r .

HOMEWORK

- Train the network:
  1) Fill in the Net class in network.py
  2) Train the network using the train.py
     - try different training parameters to train the network
  3) Plot roc curve using the script roc.py



Upload all codes and best roc curve you have got (with AP over 0.5) in brute before deadline.