

# Deep Learning

**Autonomous Robotics Lab**

Contact: [salanvoj@fel.cvut.cz](mailto:salanvoj@fel.cvut.cz)

# Timeline

- Where we will do it
- How we will do it
- What we will do

# 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

# 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	51%		N/A
43%	64C	P2	310W / 250W	10366MiB / 11178MiB		Default	
1	GeForce GTX 108...	On	00000000:05:00.0	Off	40%		N/A
52%	68C	P2	208W / 250W	8227MiB / 11178MiB		Default	
2	GeForce GTX 108...	On	00000000:08:00.0	Off	0%		N/A
29%	37C	P8	15W / 250W	0MiB / 11178MiB		Default	
3	GeForce GTX 108...	On	00000000:09:00.0	Off	0%		N/A
29%	31C	P8	15W / 250W	0MiB / 11178MiB		Default	
4	GeForce GTX 108...	On	00000000:84:00.0	Off	0%		N/A
29%	31C	P8	15W / 250W	0MiB / 11178MiB		Default	
5	GeForce GTX 108...	On	00000000:85:00.0	Off	0%		N/A
29%	35C	P8	15W / 250W	0MiB / 11178MiB		Default	
6	GeForce GTX 108...	On	00000000:88:00.0	Off	0%		N/A
29%	32C	P8	14W / 250W	0MiB / 11178MiB		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 or class desktops

- Load singularity image

```
singularity shell --bind /opt/torchenv --bind /opt/barbie /opt/ros-kinetic-desktop-full.simg
```

- Source ROS

```
source /opt/ros/kinetic/setup.bash
```

- Activate virtual environment

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

# PyTorch

## **Tensors**

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

```
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

# Linear regressor

```
import numpy as np
import torch

X = np.random.rand(30, 1)*2.0
w = np.random.rand(2, 1)
y = X*w[0] + w[1] + np.random.randn(30, 1) * 0.05
print('target w {} b {}'.format(w[0], w[1]))
Xt = torch.from_numpy(X).float()
yt = torch.from_numpy(y).float()
W = torch.rand(1, requires_grad=True)
b = torch.rand(1, requires_grad=True)

lr = 0.005
for epoch in range(2500):
    y_pred = torch.add(torch.mul(W,Xt), b) #  $W \cdot x + b$ 
    loss = torch.mean((y_pred - yt) ** 2)
    loss.backward()
    W.data = W.data - lr*W.grad.data
    b.data = b.data - lr*b.grad.data
    W.grad.data.zero_()
    b.grad.data.zero_()

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

# Imports

# Create data as numpy arrays

# Randomly select weights of linear regressor

# Create targets

# Convert numpy arrays to torch tensors

# Initialize weights randomly with parameter requires\_grad=True

# set up learning rate

# Compute predictions

# Compute cost function

# Run back-propagation

# Update parameters

# Reset gradients

# Linear regressor

```

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 of linear regressor
y = X*w[0] + w[1] + np.random.randn(30, 1) * 0.05  # Create targets
print('target w {} b {}'.format(w[0], w[1]))
Xt = torch.from_numpy(X).float()  # Convert numpy arrays to torch tensors
yt = torch.from_numpy(y).float()

W = torch.rand(1, requires_grad=True)  # Initialize weights randomly with parameter requires_grad=True
b = torch.rand(1, requires_grad=True)

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

print('found w {} b {}'.format(W.data ,b.data))

```

# Linear regressor

```

import numpy as np
import torch

X = np.random.rand(30, 1)*2.0
w = np.random.rand(2, 1)
y = X*w[0] + w[1] + np.random.randn(30, 1) * 0.05
print('target w {} b {}'.format(w[0], w[1]))
Xt = torch.from_numpy(X).float()
yt = torch.from_numpy(y).float()
W = torch.rand(1, requires_grad=True)
b = torch.rand(1, requires_grad=True)

lr = 0.005
for epoch in range(2500):
    y_pred = torch.add(torch.mul(W,Xt), b) # W*x + b
    loss = torch.mean((y_pred - yt) ** 2)
    loss.backward()
    W.data = W.data - lr*W.grad.data
    b.data = b.data - lr*b.grad.data
    W.grad.data.zero_()
    b.grad.data.zero_()

print('found w {} b {}'.format(W.data ,b.data))

```

# Imports

# Create data as numpy arrays

# Randomly select weights of linear regressor

# Create targets

# Convert numpy arrays to torch tensors

# Initialize weights randomly with parameter requires\_grad=True

# set up learning rate

# Compute predictions

# Compute cost function

# Run back-propagation

# Update parameters

# Reset gradients

# Linear regressor

```

import numpy as np
import torch

X = np.random.rand(30, 1)*2.0
w = np.random.rand(2, 1)
y = X*w[0] + w[1] + np.random.randn(30, 1) * 0.05
print('target w {} b {}'.format(w[0], w[1]))
Xt = torch.from_numpy(X).float()
yt = torch.from_numpy(y).float()
W = torch.rand(1, requires_grad=True)
b = torch.rand(1, requires_grad=True)

lr = 0.005
for epoch in range(2500):
    y_pred = torch.add(torch.mul(W,Xt), b) # W*x + b
    loss = torch.mean((y_pred - yt) ** 2)
    loss.backward()
    W.data = W.data - lr*W.grad.data
    b.data = b.data - lr*b.grad.data
    W.grad.data.zero_()
    b.grad.data.zero_()

print('found w {} b {}'.format(W.data ,b.data))

```

# Imports

# Create data as numpy arrays

# Randomly select weights of linear regressor

# Create targets

# Convert numpy arrays to torch tensors

# Initialize weights randomly with parameter requires\_grad=True

# set up learning rate

# Compute predictions

# Compute cost function

# Run back-propagation

# Update parameters

# Reset gradients

# Linear regressor

```

import numpy as np
import torch

X = np.random.rand(30, 1)*2.0
w = np.random.rand(2, 1)
y = X*w[0] + w[1] + np.random.randn(30, 1) * 0.05
print('target w {} b {}'.format(w[0], w[1]))
Xt = torch.from_numpy(X).float()
yt = torch.from_numpy(y).float()
W = torch.rand(1, requires_grad=True)
b = torch.rand(1, requires_grad=True)

lr = 0.005
for epoch in range(2500):
    y_pred = torch.add(torch.mul(W,Xt), b) # W*x + b
    loss = torch.mean((y_pred - yt) ** 2)
    loss.backward()
    W.data = W.data - lr*W.grad.data
    b.data = b.data - lr*b.grad.data
    W.grad.data.zero_()
    b.grad.data.zero_()

print('found w {} b {}'.format(W.data ,b.data))

```

# Imports

# Create data as numpy arrays

# Randomly select weights of linear regressor

# Create targets

# Convert numpy arrays to torch tensors

# Initialize weights randomly with parameter requires\_grad=True

# set up learning rate

# Compute predictions

# Compute cost function

# Run back-propagation

# Update parameters

# Reset gradients

# Linear regressor

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
```

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



# Barbie detector

- Our goal is to implement program that will detect barbie in image and gives us a 3d coordinates of that barbie.

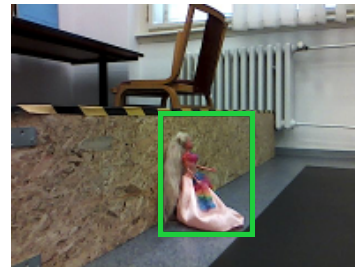
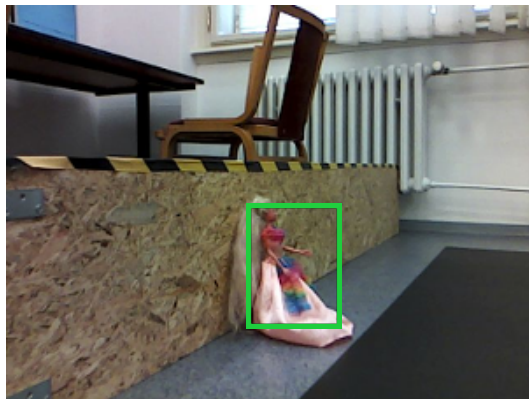
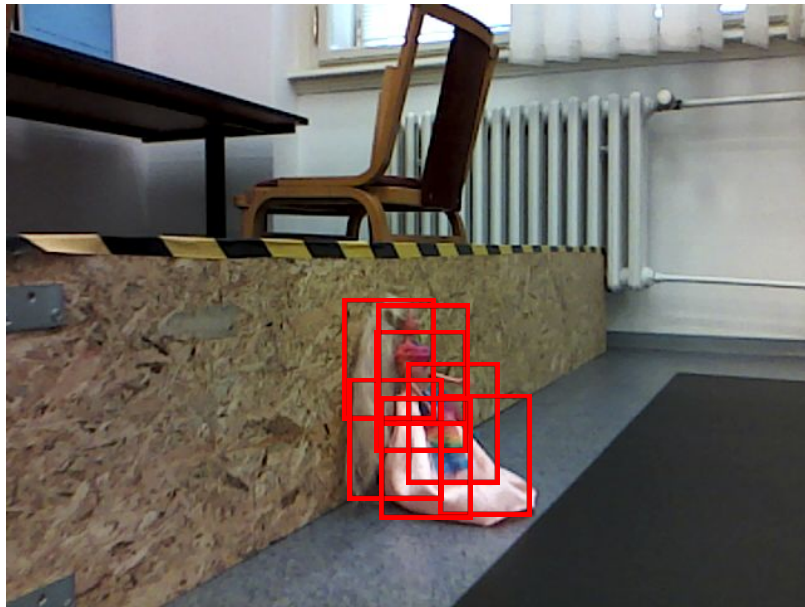


X, Y, Z, according to robot

# Model

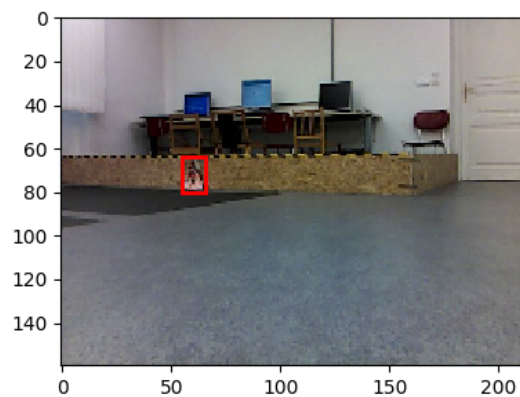
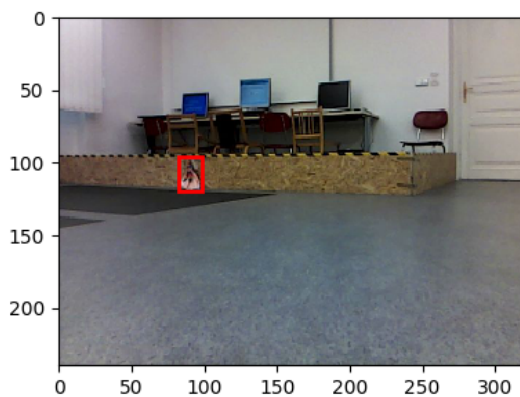
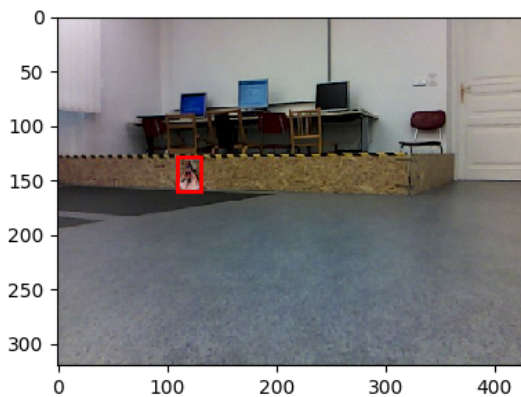
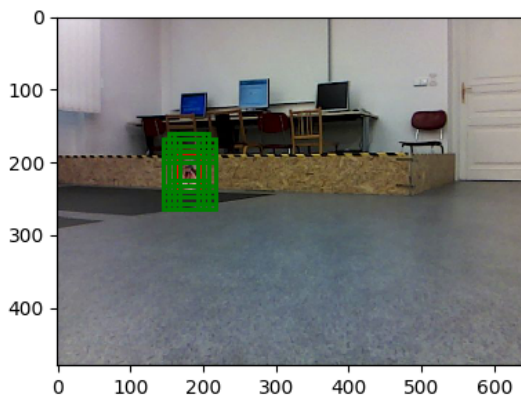
- We use feature extractor part of the network called YOLO V2 TINY
- We run the network on the multiple input image scales to detect barbies of all sizes

Network will find barbies size are near to 64x48 pixels

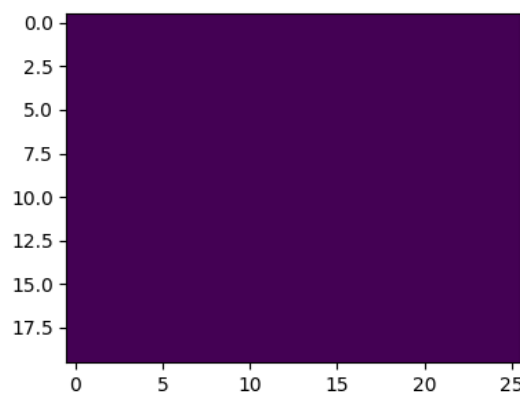
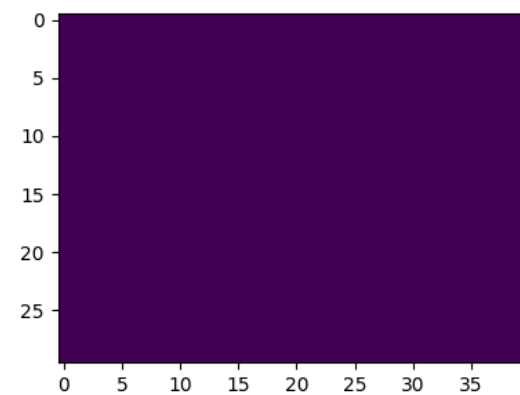
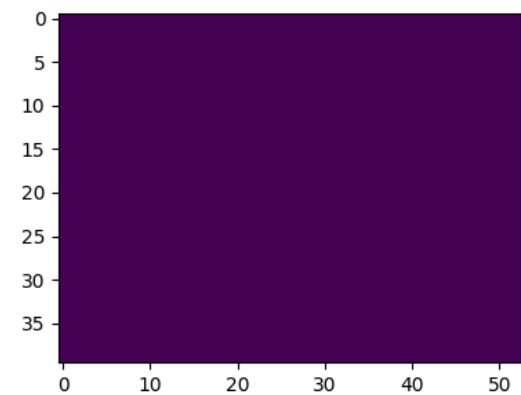
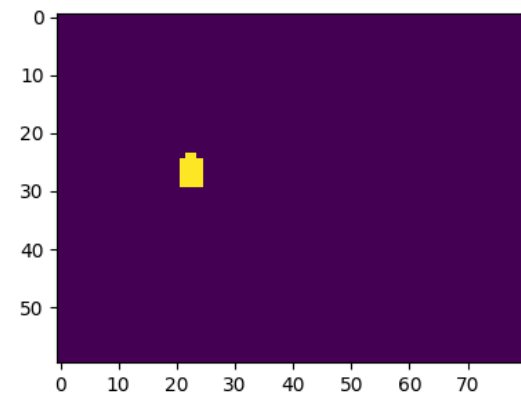


# Training dataset

Input

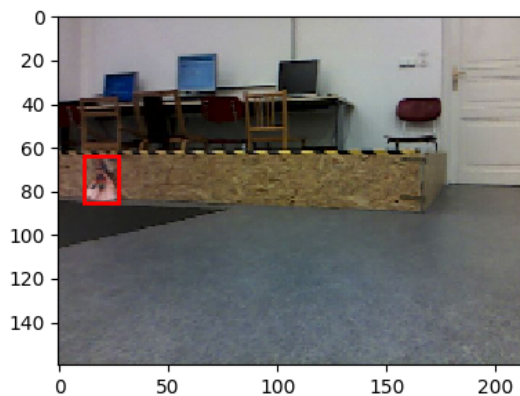
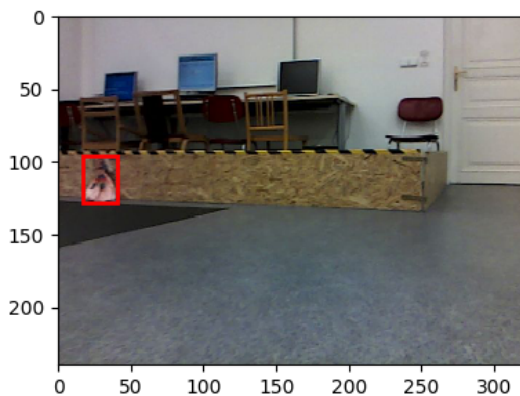
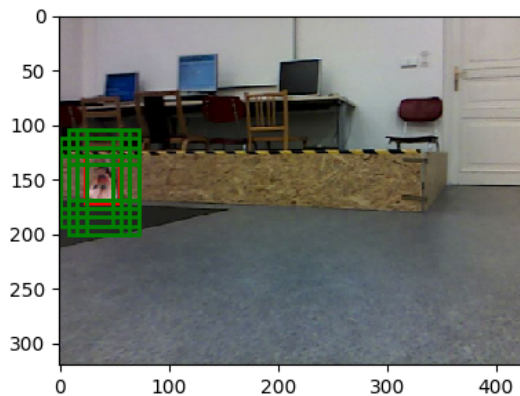
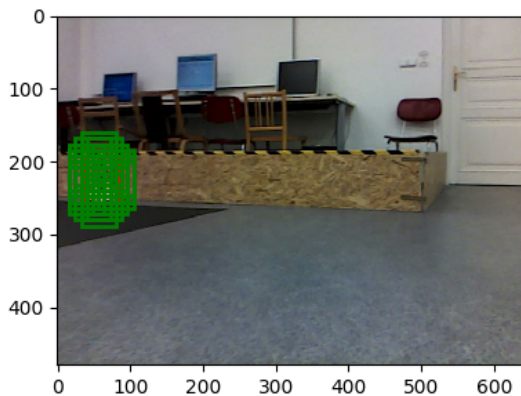


Label

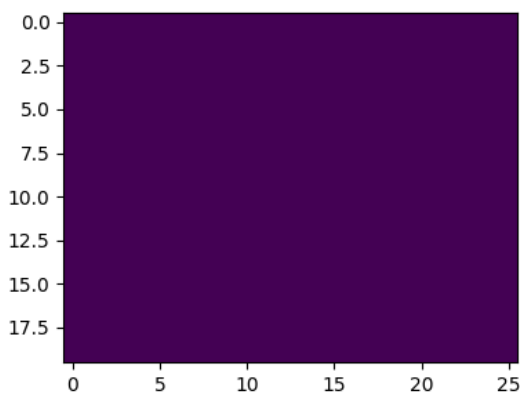
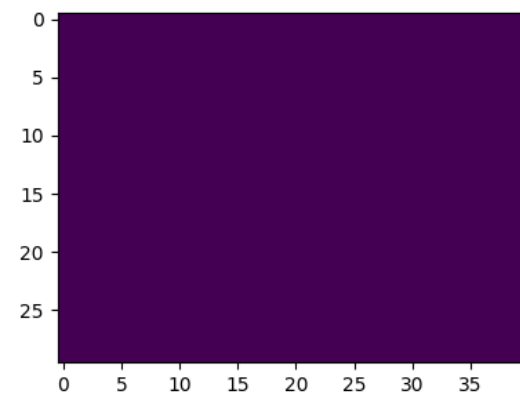
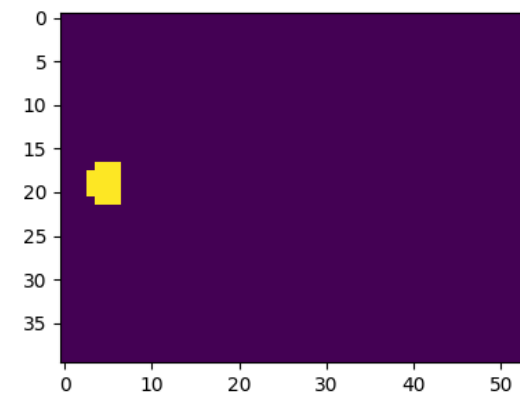
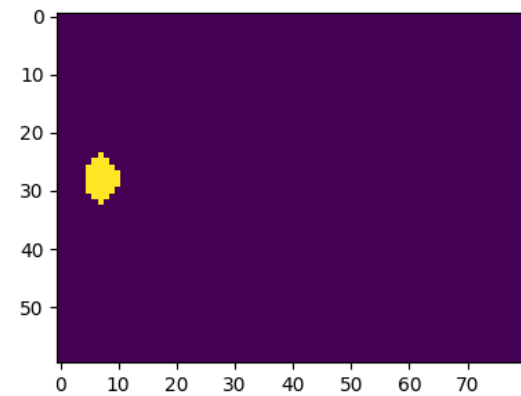


# Training dataset

Input

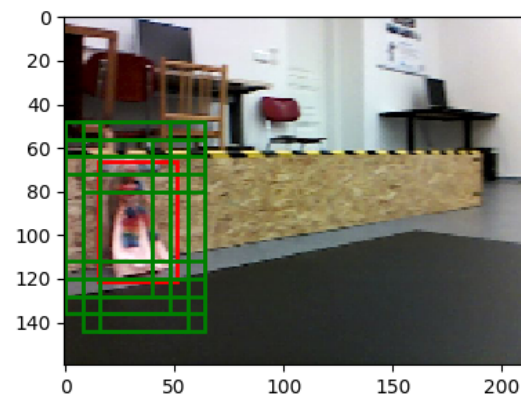
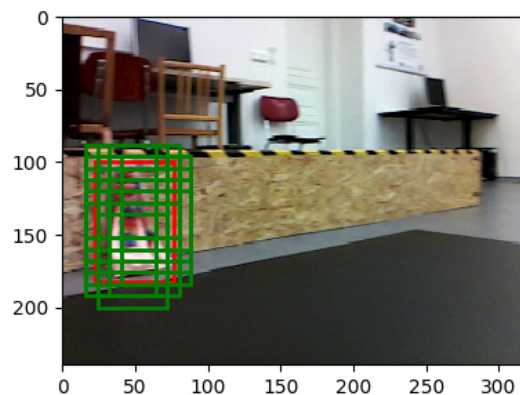
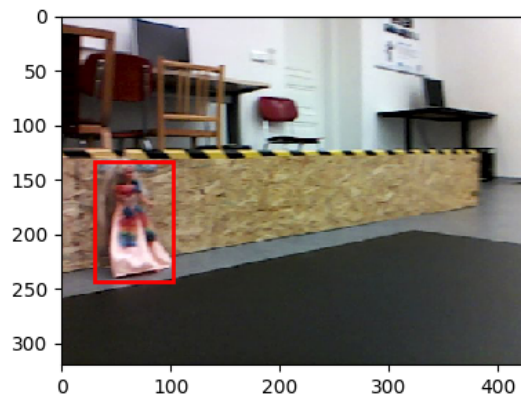
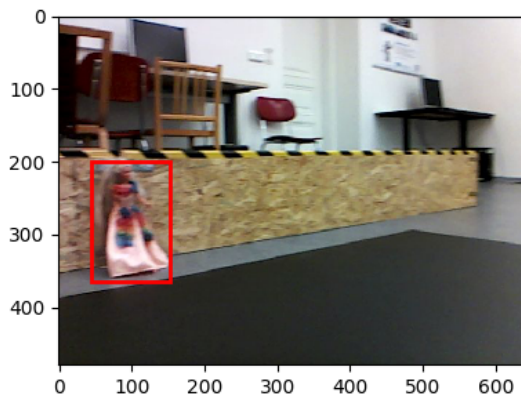


Label

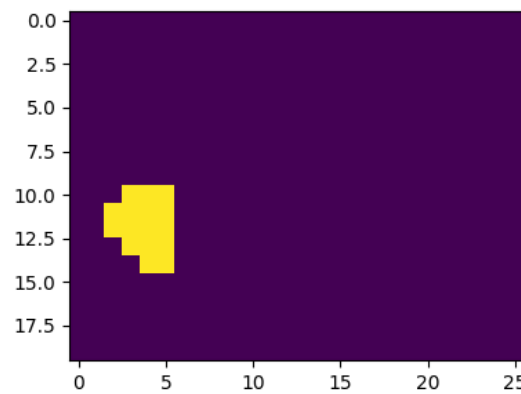
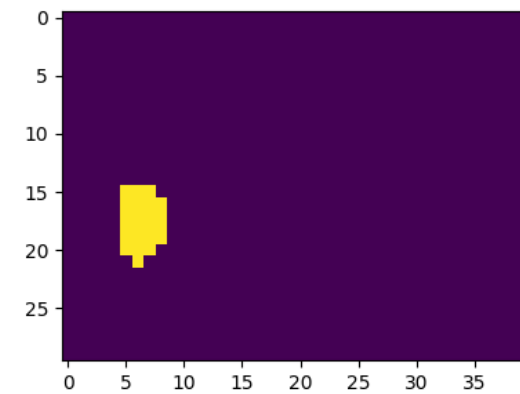
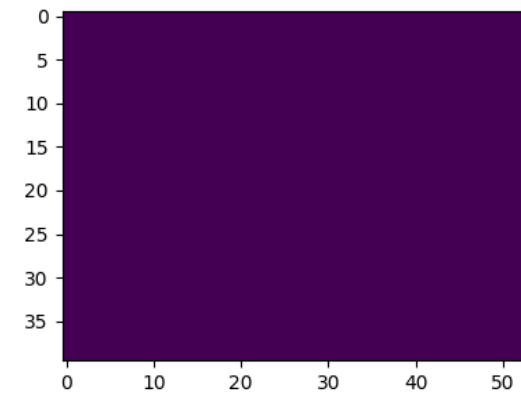
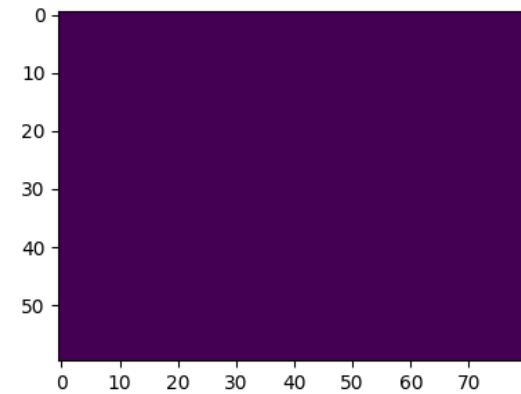


# Training dataset

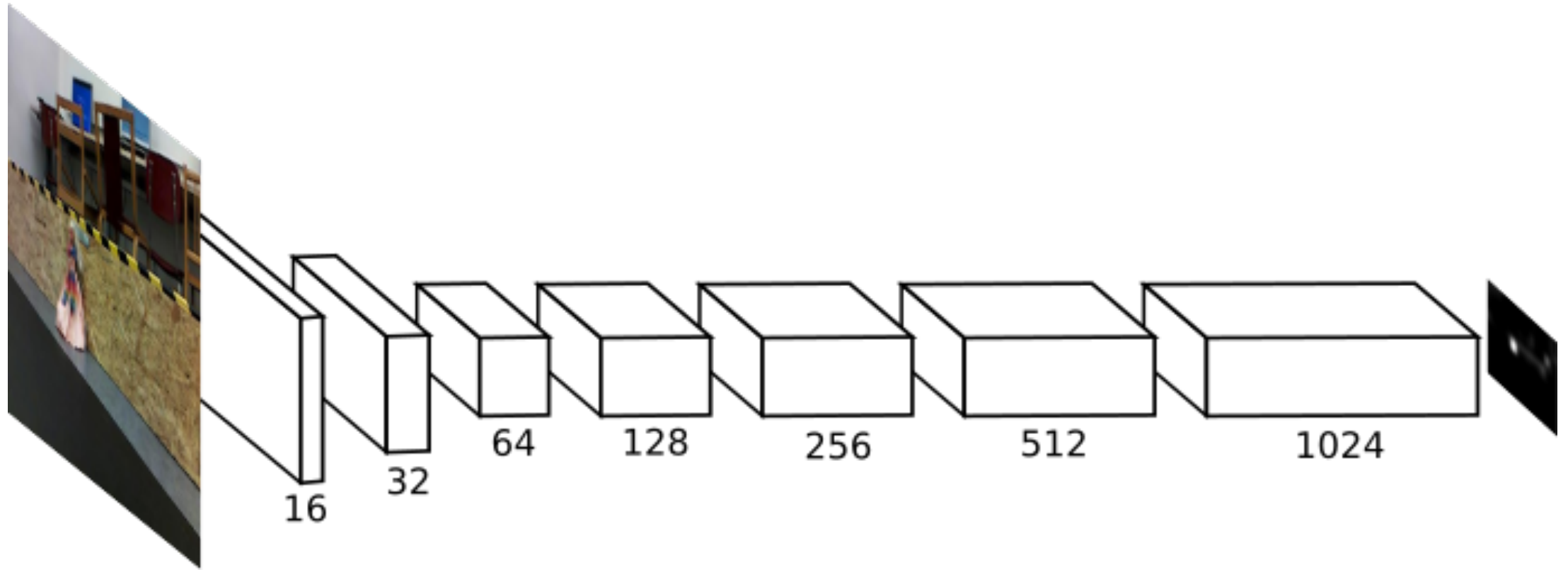
Input



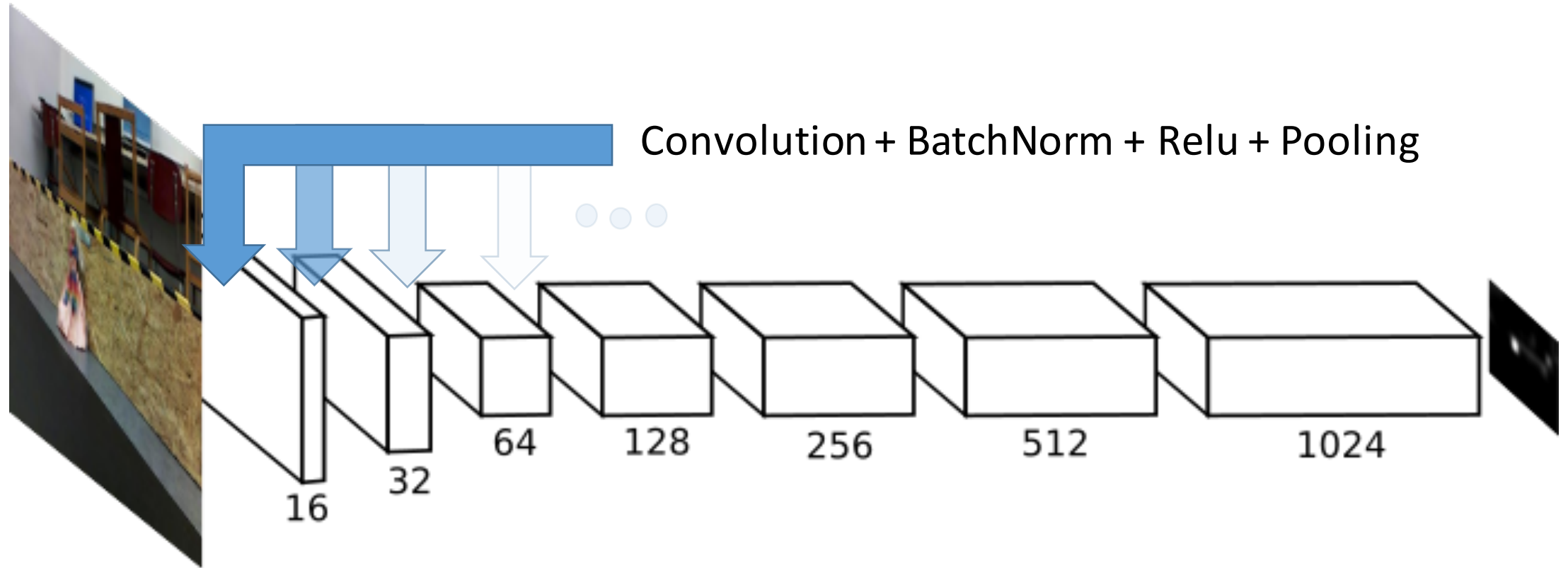
Label



# 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 = nn.LeakyReLU(0.1, inplace=True)
    self.pool1 = nn.MaxPool2d(2, 2)
    self.pool2 = nn.MaxPool2d(3, 1, 1)
    self.conv1 = nn.Conv2d(3, 16, 3, 1, 1, bias=False)
    self.bn1 = nn.BatchNorm2d(16)
    self.conv2 = nn.Conv2d(16, 32, 3, 1, 1, bias=False)
    self.bn2 = nn.BatchNorm2d(32)
    self.conv3 = nn.Conv2d(32, 64, 3, 1, 1, bias=False)
    self.bn3 = nn.BatchNorm2d(64)
    self.conv4 = nn.Conv2d(64, 128, 3, 1, 1, bias=False)
    self.bn4 = nn.BatchNorm2d(128)
    self.conv5 = nn.Conv2d(128, 256, 3, 1, 1, bias=False)
    self.bn5 = nn.BatchNorm2d(256)
    self.conv6 = nn.Conv2d(256, 512, 3, 1, 1, bias=False)
    self.bn6 = nn.BatchNorm2d(512)
    self.conv7 = nn.Conv2d(512, 1024, 3, 1, 1, bias=False)
    self.bn7 = nn.BatchNorm2d(1024)
    self.conv8 = nn.Conv2d(1024, 1, 1, 1, 1)
```

```
def forward(self, input):
```

```
    x = self.pool1(self.relu(self.bn1(self.conv1(input))))
```

```
    x = self.pool1(self.relu(self.bn2(self.conv2(x))))
```

```
    x = self.pool1(self.relu(self.bn3(self.conv3(x))))
```

```
    x = self.pool2(self.relu(self.bn4(self.conv4(x))))
```

```
    x = self.pool2(self.relu(self.bn5(self.conv5(x))))
```

```
    x = self.pool2(self.relu(self.bn6(self.conv6(x))))
```

```
    x = self.relu(self.bn7(self.conv7(x)))
```

```
    x = (self.conv8(x))
```

```
    return x
```

# Training script

- Training script is prepared for you!

You only need to set paths for training and validation data

and set training variables properly:

`batch_size`

`number of epochs`

`learning_rate`

`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
    - (use ssh with +X parameter to allow graphic)

DEADLINE next week! (1. 4. - 4. 4.)

