Deep Learning Using PYTORCH

Masoud Pourreza

Pourreza.masoud@gmail.com

https://www.aparat.com/partdpai

مرکز تحقیقات هوش پارت

Torch

- Open source machine learning library
- Scientific computing framework
- Based on the Lua !!
- Research Companies and Labs
- Facebook AI research, Google + Deep mind , Twitter, NVIDIA
- Started at 2002
- website

PyTorch

- PyTorch is a python package that provides two high-level features:
  - Tensor computation (like numpy) with strong GPU acceleration
  - Deep Neural Networks built on a tape-based autograd system
- PyTorch provides Tensors that can live either on the CPU or the GPU
- PyTorch is a python library
- More Pythonic (imperative)
- Flexible
- Intuitive and cleaner code
- Easy to debug
- More Neural Networkic
- Write code as the network works
- forward/backward

PyTorch packages

Package	Description
torch	a Tensor library like NumPy, with strong GPU support
torch.autograd	a tape based automatic differentiation library that supports all differentiable Tensor operations in torch
torch.nn	a neural networks library deeply integrated with autograd designed for maximum flexibility
torch.optim	an optimization package to be used with torch.nn with standard optimization methods such as SGD, RMSProp, LBFGS, Adam etc.
torch.multiprocessing	python multiprocessing, but with magical memory sharing of torch Tensors across processes. Useful for data loading and hogwild training.
torch.utils	DataLoader, Trainer and other utility functions for convenience
torchvision	The torchvision package consists of popular datasets, model architectures, and common image transformations for computer vision.

Installation requirements

Python on ubuntu

Pip on ubuntu

conda on ubuntu

Python on windows

Pip on windows

conda on windows

PyTorch Installation

http://pytorch.org

Get Started.

Select your preferences, then run the PyTorch install command.

Please ensure that you are on the latest pip and numpy packages.
Anaconda is our recommended package manager

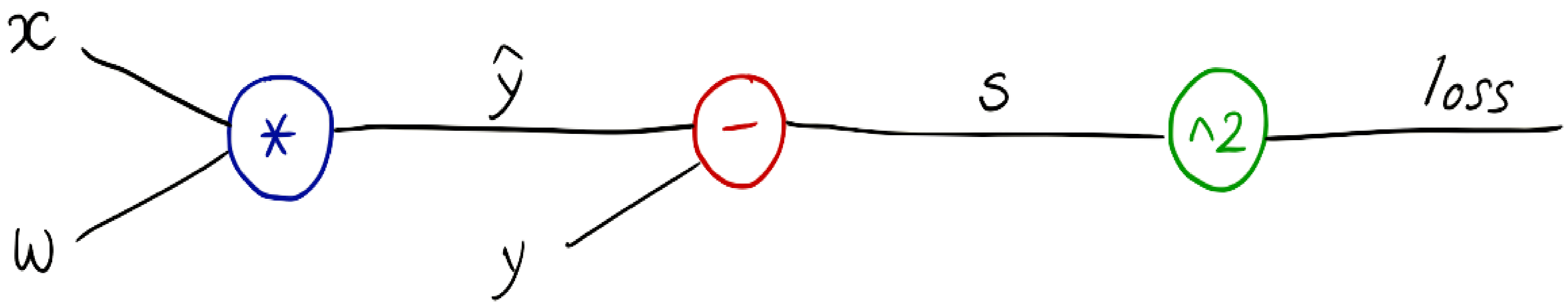
OS	Linux	MacOS	Windows	
Package Manager	conda	pip	Source	
Python	2.7	3.5	3.6	3.7
CUDA	8	9.0	9.2	None

Run this command: pip3 install torch torchvision

Click here for previous versions of PyTorch

Computational Graph

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$



PyTorch Basics

Classification Problem

airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck

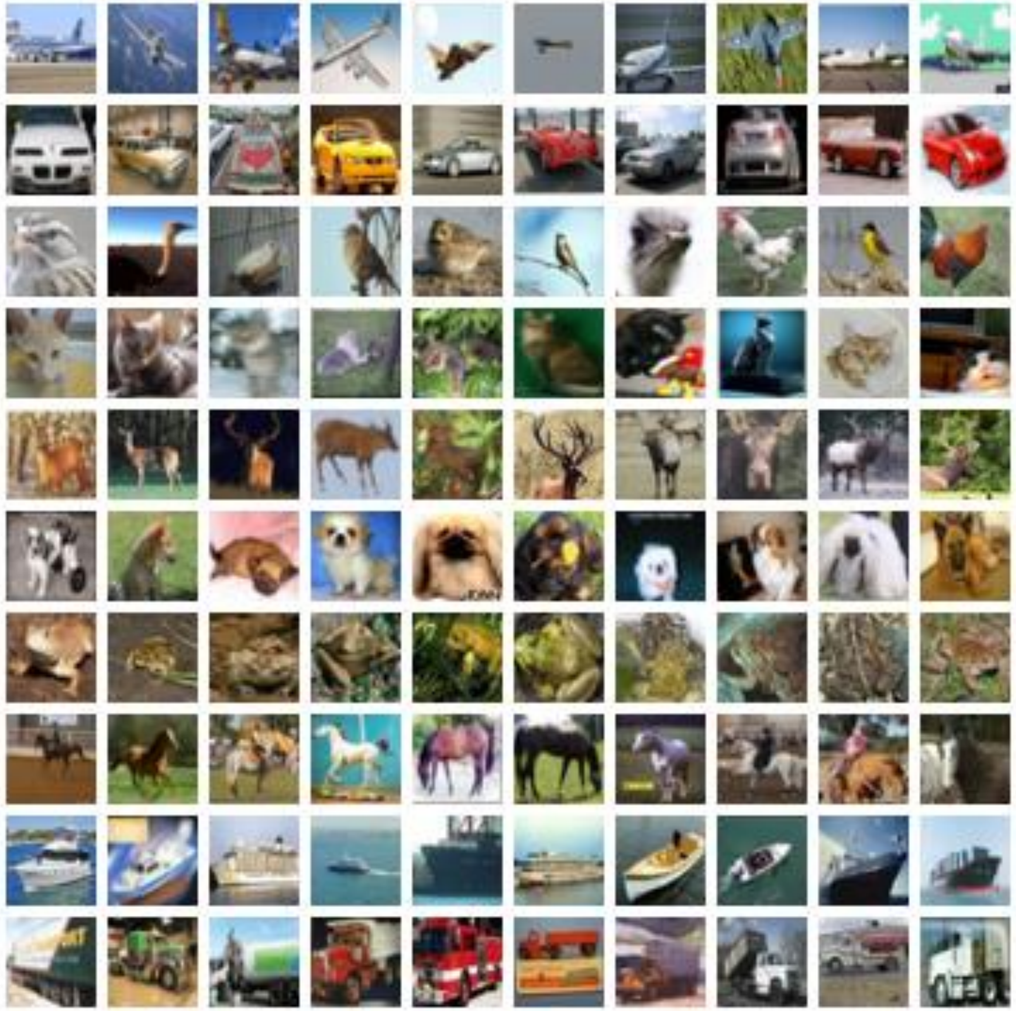


Image Size = 3×32×32

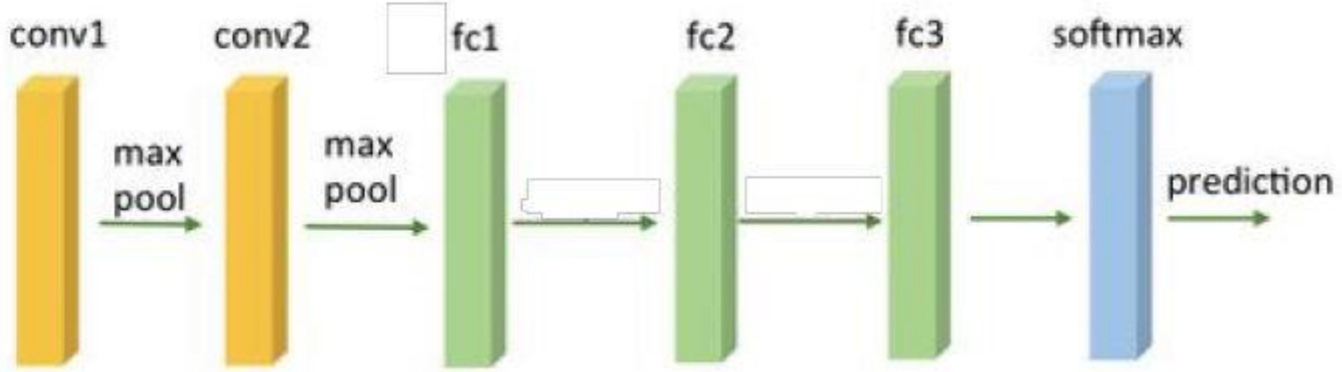
PyTorch Rhythm

1. Design your model using class with Variables
2. Construct loss and optimizer (select from PyTorch API)
3. Training cycle (forward, backward, update)

Classification Problem

1 **Design your model using class with Variables**

```
##########################################################################
# 1. Define a Neural Network
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Copy the neural network from the Neural Networks section before and modify it to
# take 3-channel images (instead of 1-channel images as it was defined).

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()

##########################################################################
```



Cifar10 classification

❷ Construct loss and optimizer (select from PyTorch API)

```
########################################################################
# 2. Define a Loss function and optimizer
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Let's use a Classification Cross-Entropy loss and SGD with momentum
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

########################################################################
```

Cifar10 classification

```
########################################################################
# 3. Train the network
# ^^^^^^^^^^^^^^^^^^^^
#
# This is when things start to get interesting.
# We simply have to loop over our data iterator, and feed the inputs to the
# network and optimize
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # wrap them in Variable
        inputs, labels =         (inputs),          labels)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.data[0]
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

3 Training cycle (forward, backward, update)

Batch (batch size)

```python
# Training cycle
for epoch in range(training_epochs):
    # Loop over all batches
    for i in range(total_batch):
        batch_xs, batch_ys = …
```

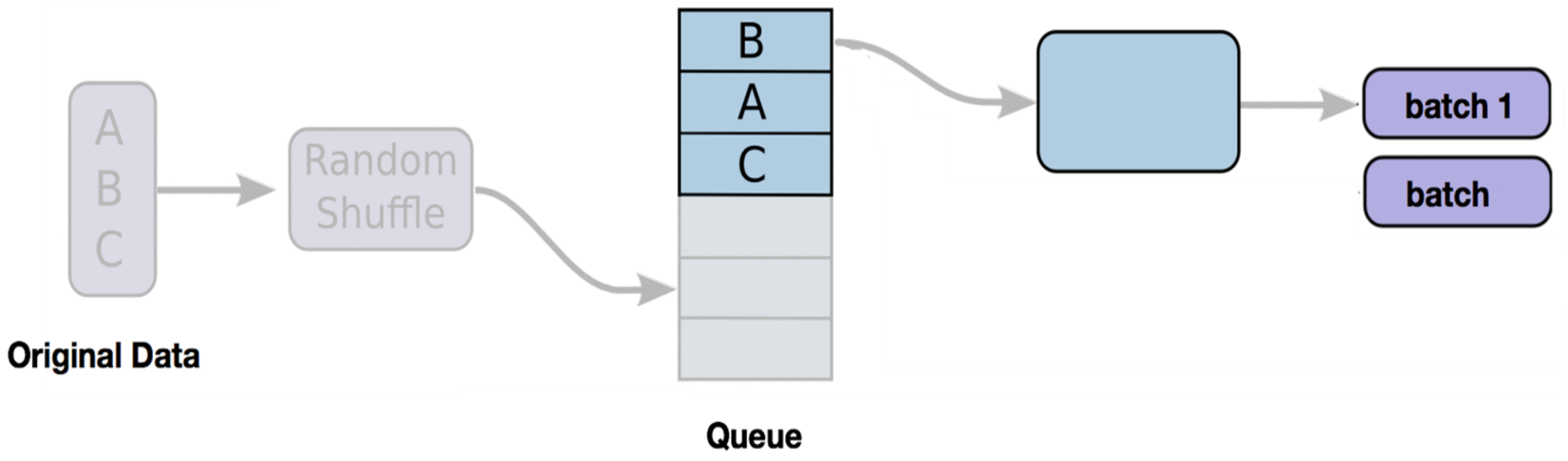
288

In the neural network terminology:

- one **epoch** = one forward pass and one backward pass of *all* the training examples
- **batch size** = the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need.
- number of **iterations** = number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

Example: if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.

Data Loader

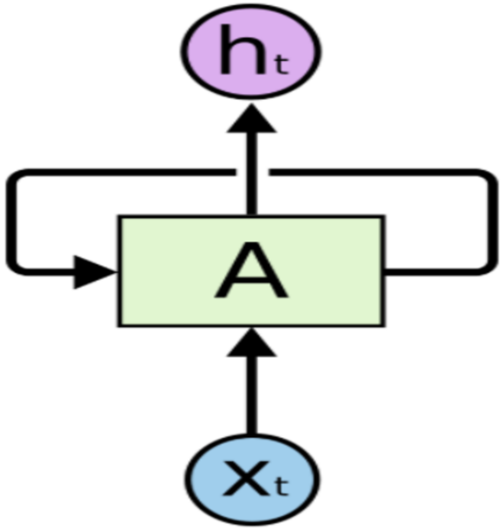


Transfer Learning / freeze layers

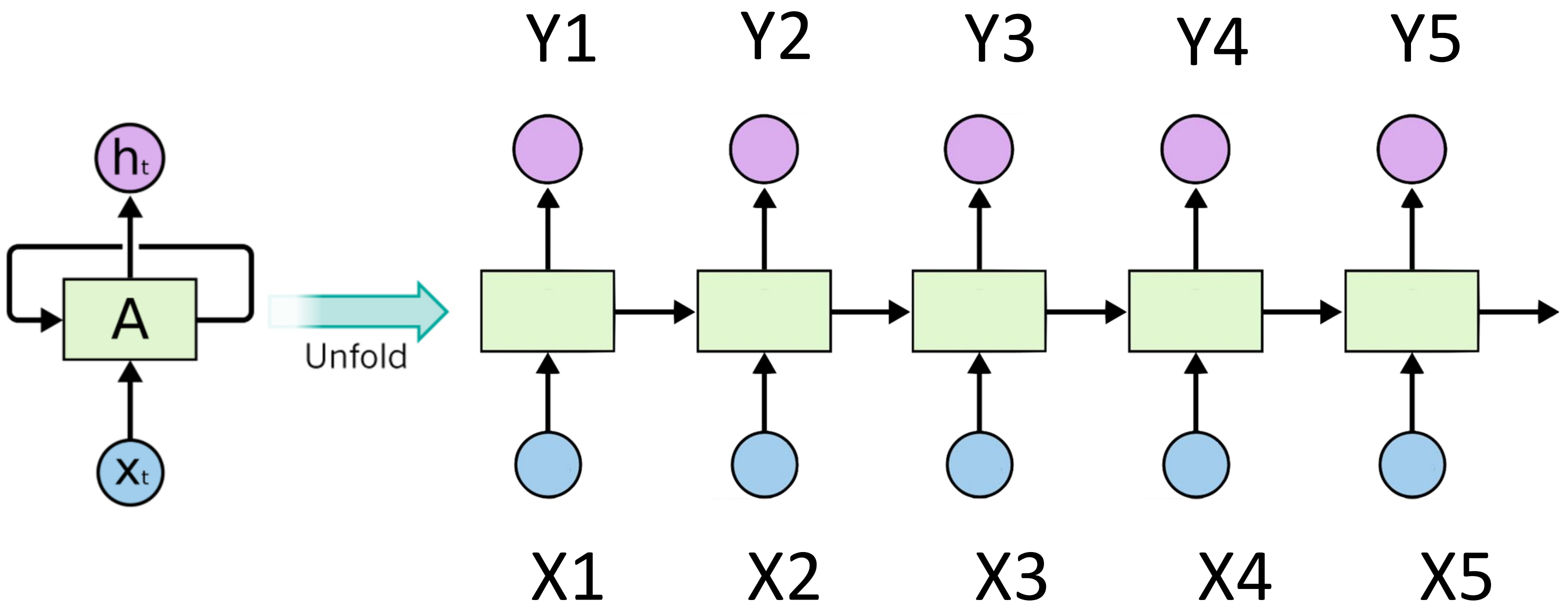
- Transfer learning is a machine learning technique where a model trained on one task is re-purposed on a second related task.

- Transfer learning is an optimization that allows rapid progress or improved performance when modeling the second task

RNN in Pytorch



RNN

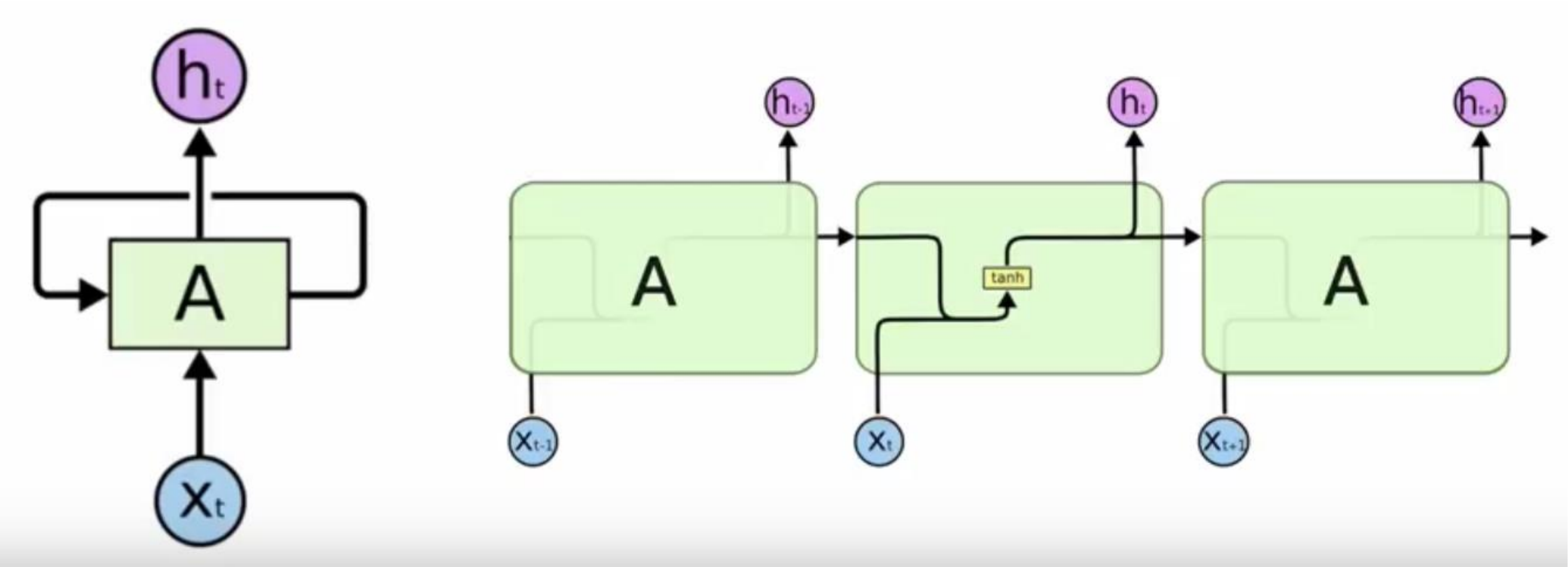


مرکز تحقیقات هوش پارت

RNN Applications: series of data

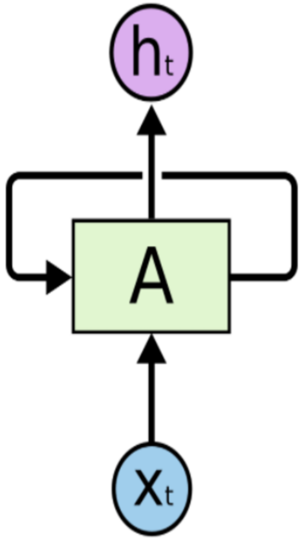
- Time series prediction
- Language modeling (text generation)
- Text sentiment analysis
- Named entity recognition
- Translation
- Speech recognition
- Anomaly detection in time series
- Music composition
- ...

RNN inside



RNN in PyTorch

```
cell = nn.RNN(input_size=4, hidden_size=2, batch_first=True)
cell = nn.LSTM(input_size=4, hidden_size=2, batch_first=True)
cell = nn.GRU(input_size=4, hidden_size=2, batch_first=True)
```



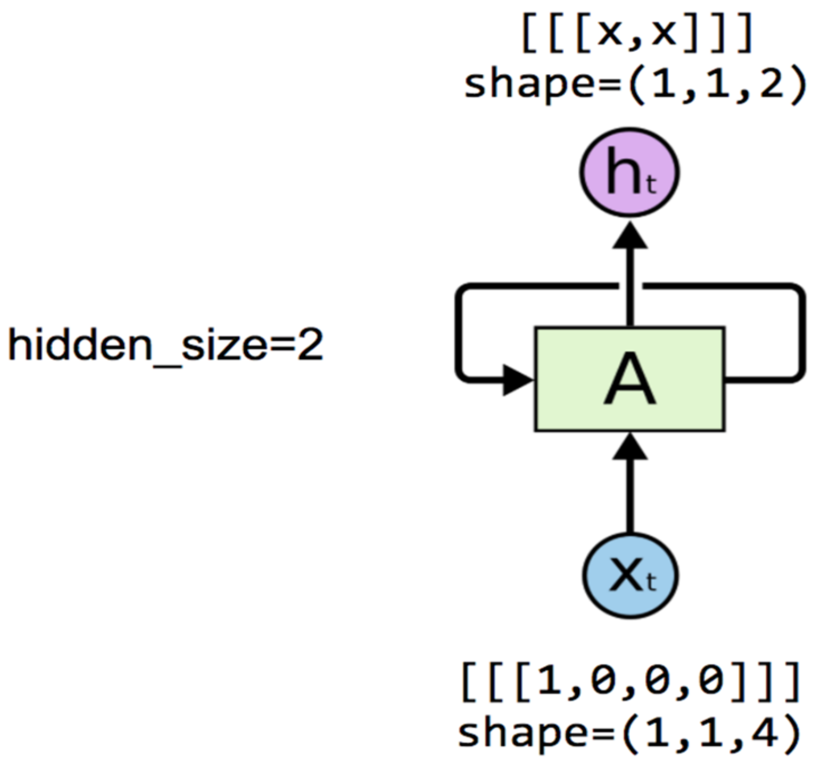

```
inputs = ... # (batch_size, seq_len, input_size) with batch_first=True
hidden = (... , ...) # (num_layers, batch_size, hidden_size)
```

```
out, hidden = cell(inputs, hidden)
```

One hot encoding for letters, h, e, l, l, o

```
# One hot encoding
h = [1, 0, 0, 0]
e = [0, 1, 0, 0]
l = [0, 0, 1, 0]
o = [0, 0, 0, 1]
```

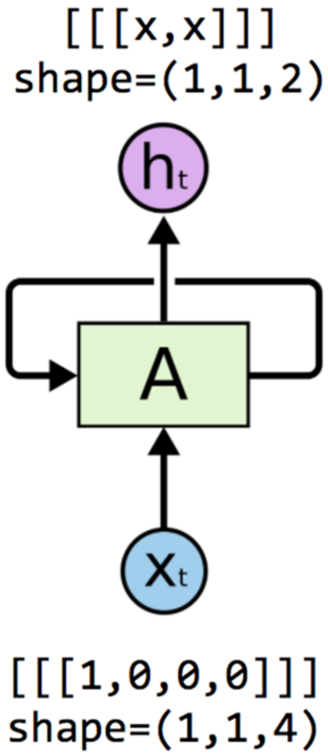
One node: 4 (*input-dim*) in 2 (*hidden_size*)




```
# One hot encoding
h = [1, 0, 0, 0]
e = [0, 1, 0, 0]
l = [0, 0, 1, 0]
o = [0, 0, 0, 1]
```

One node: 4 (*input-dim*) in 2 (*hidden_size*)

hidden_size=2



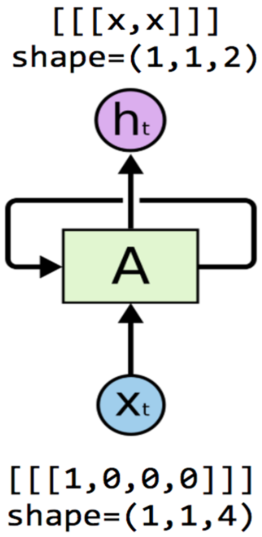

```
# One hot encoding
h = [1, 0, 0, 0]
e = [0, 1, 0, 0]
l = [0, 0, 1, 0]
o = [0, 0, 0, 1]
```

One node: 4 (*input_dim*) in 2 (*hidden_size*)

```
# One cell RNN input_dim (4) -> output_dim (2)
cell = nn.LSTM(input_size=4, hidden_size=2, batch_first=True)

# One letter input
inputs = (torch.Tensor([[h]])) # rank = (1, 1, 4)

# initialize the hidden state.
# (num_layers * num_directions, batch, hidden_size)
hidden = ((torch.randn(1, 1, 2)))

# Feed to one element at a time.
# after each step, hidden contains the hidden state.
out, hidden = cell(inputs, hidden)
print("out", out.data)
```

```
-0.1243  0.0738
[torch.FloatTensor of size 1x1x2]
```

hidden_size=2

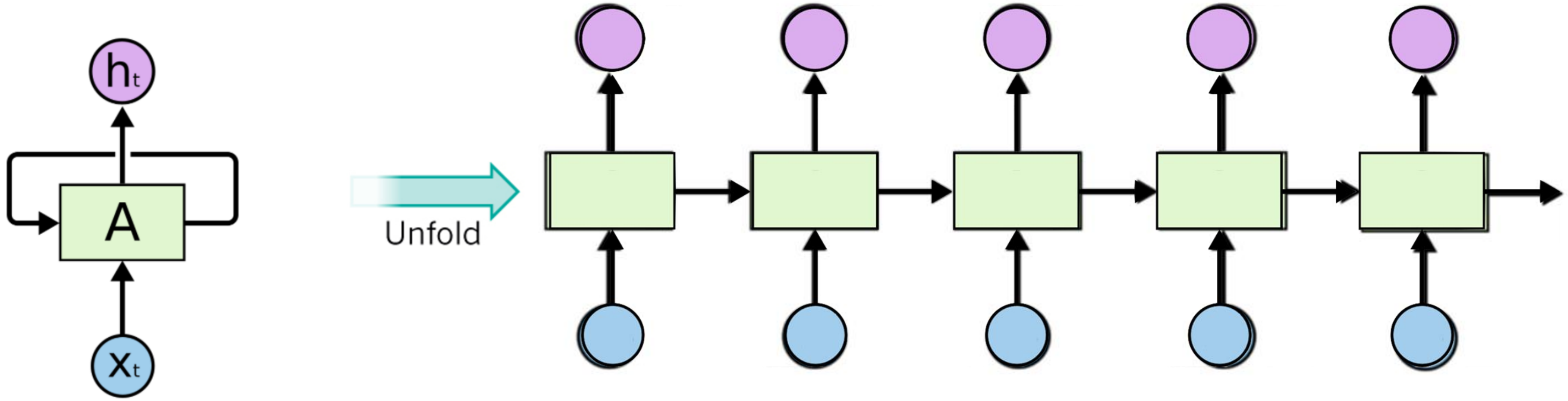



```
# One hot encoding
h = [1, 0, 0, 0]
e = [0, 1, 0, 0]
l = [0, 0, 1, 0]
o = [0, 0, 0, 1]
```

Unfolding to n sequences

```
hidden_size=2
seq_len=5
```

shape=(1,5,2): [[[x,x], [x,x], [x,x], [x,x], [x,x]]]



shape=(1,5,4): [[[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,1,0], [0,0,0,1]]]
h e l l o

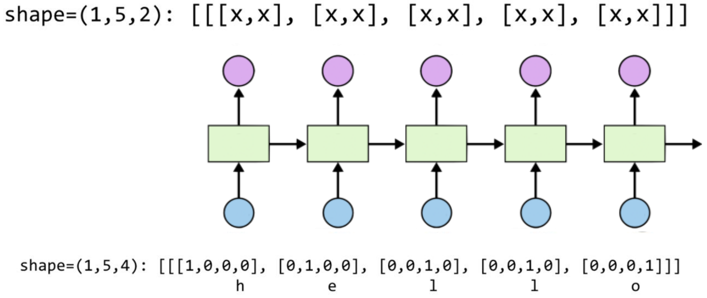
Unfolding to n sequences

```python
# One cell RNN input_dim (4) -> output_dim (2). sequence: 5
cell = nn.LSTM(input_size=4, hidden_size=2, batch_first=True)

inputs = (torch.Tensor([[h, e, l, l, o]]))
print("input size", inputs.size())

hidden = ((torch.randn(1, 1, 2)))  # clean out hidden state

out, hidden = cell(inputs, hidden)
print(out.data)
```

```python
# One hot encoding
h = [1, 0, 0, 0]
e = [0, 1, 0, 0]
l = [0, 0, 1, 0]
o = [0, 0, 0, 1]
```

```
input size torch.Size([1, 5, 4])

(0 ,.,.) =
 -0.1825  0.0737
 -0.1981  0.1164
 -0.3367  0.2095
 -0.3625  0.2503
 -0.2038  0.3626
[torch.FloatTensor of size 1x5x2]
```

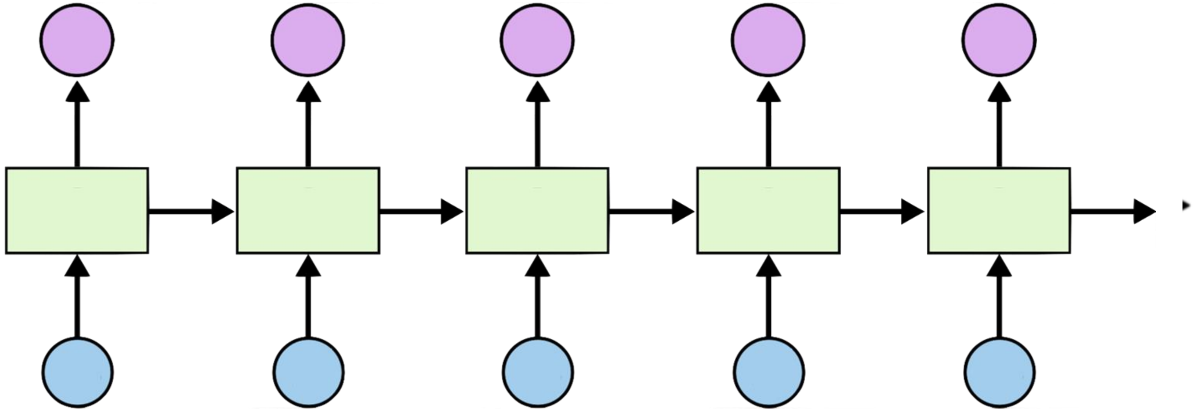
```
hidden_size=2
seq_len=5
```



```
Hidden_size=2
sequence_length=5
batch_size=3
```

Batching input

```
shape=(3,5,2): [[[x,x], [x,x], [x,x], [x,x], [x,x]],
                [[x,x], [x,x], [x,x], [x,x], [x,x]],
                [[x,x], [x,x], [x,x], [x,x], [x,x]]]
```



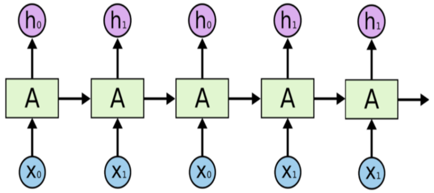
```
shape=(3,5,4): [[[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,1,0], [0,0,0,1]], # hello
                [[0,1,0,0], [0,0,0,1], [0,0,1,0], [0,0,1,0], [0,0,1,0]]  # eolll
                [[0,0,1,0], [0,0,1,0], [0,1,0,0], [0,1,0,0], [0,0,1,0]]] # lleel
```

Batching input

```
# One cell RNN input_dim (4) -> output_dim (2). sequence: 5, batch 3
# 3 batches 'hello', 'eolll', 'lleel'
# rank = (3, 5, 4)
inputs = (torch.Tensor([[h, e, l, l, o],
                        [e, o, l, l, l],
                        [l, l, e, e, l]]))
print("input size", inputs.size()) # input size torch.Size([3, 5, 4])

# (num_layers * num_directions, batch, hidden_size)
hidden = ((torch.randn(1, 3, 2)),(
   torch.randn((1, 3, 2))))

out, hidden = cell(inputs, hidden)
print("out size", out.size()) # out size torch.Size([3, 5, 2])
```

hidden_size=2
sequacne_length=5
batch = 3

```
shape=(3,5,2): [[[x,x], [x,x], [x,x], [x,x], [x,x]],
                [[x,x], [x,x], [x,x], [x,x], [x,x]],
                [[x,x], [x,x], [x,x], [x,x], [x,x]]]
```

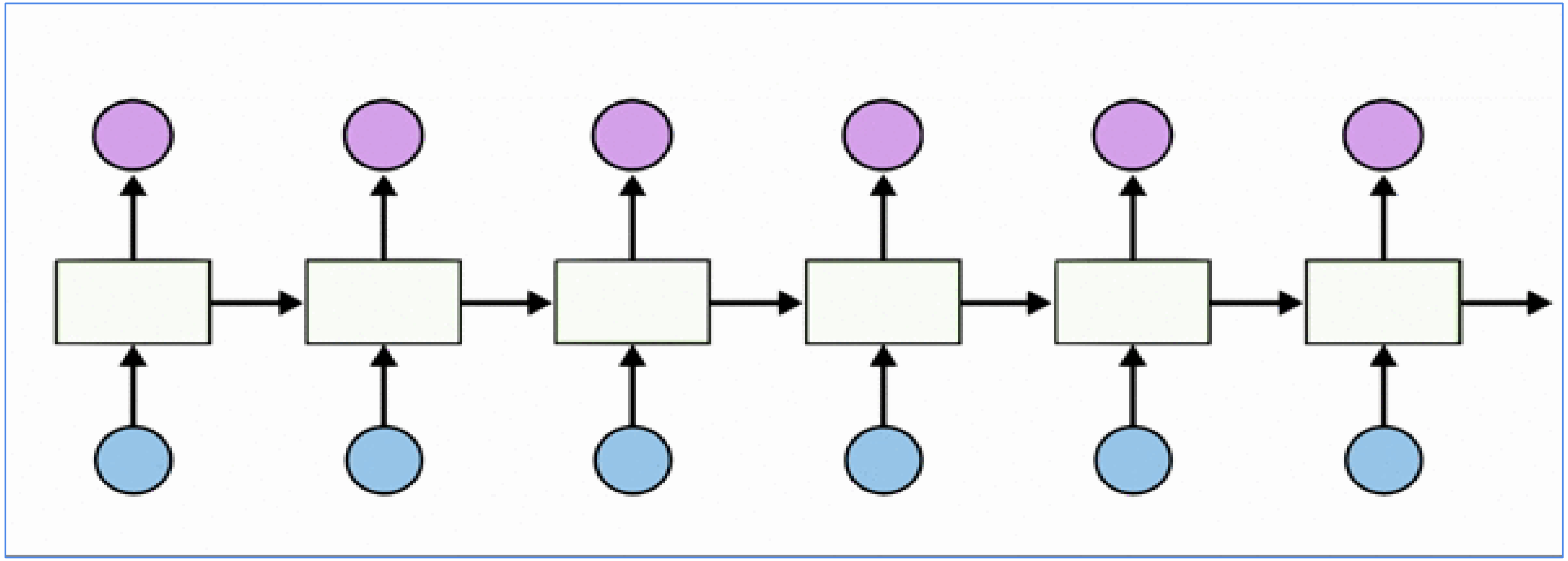



```
shape=(3,5,4): [[[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,1,0], [0,0,0,1]], # hello
                [[0,1,0,0], [0,0,0,1], [0,0,1,0], [0,0,1,0], [0,0,1,0]]  # eolll
                [[0,0,1,0], [0,0,1,0], [0,1,0,0], [0,1,0,0], [0,0,1,0]]] # lleel
```

Hidden_size=2
sequence_length=5
batch_size=3

Teach RNN 'hihell' to 'ihello'

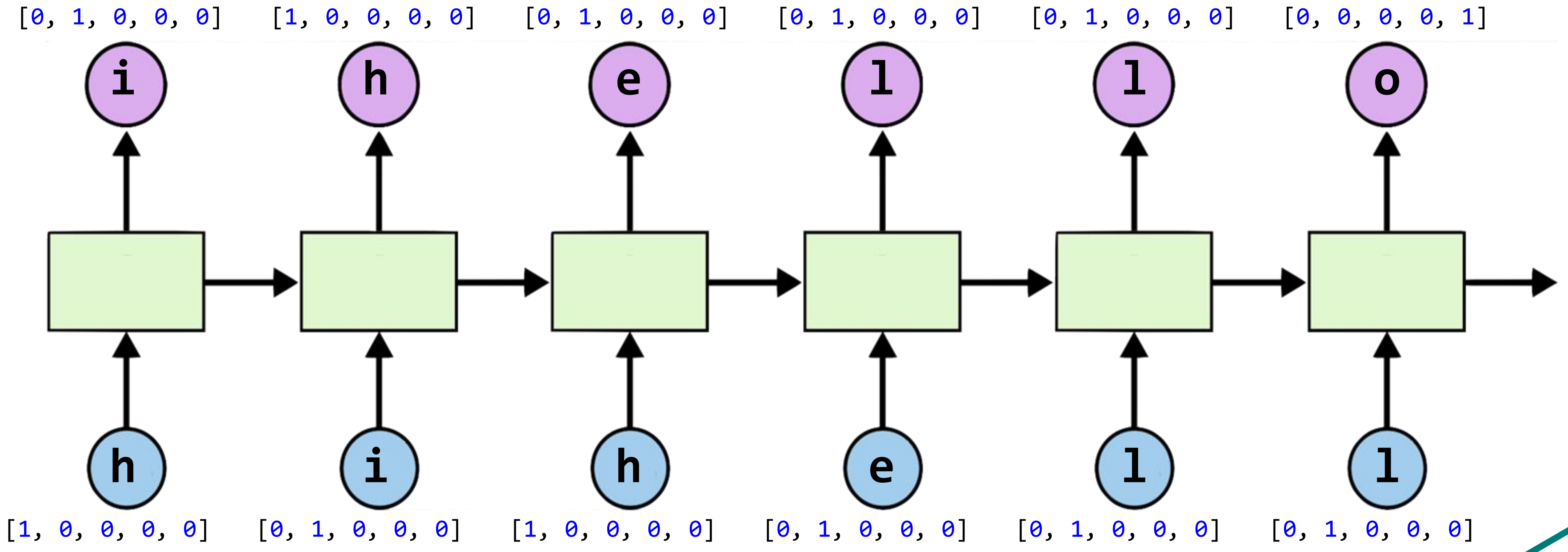
```
[1, 0, 0, 0, 0],    # h 0
[0, 1, 0, 0, 0],    # i 1
[0, 0, 1, 0, 0],    # e 2
[0, 0, 0, 1, 0],    # l 3
[0, 0, 0, 0, 1],    # o 4
```



Teach RNN ‘hihell’ to ‘ihello’

```
[1, 0, 0, 0, 0],    # h 0
[0, 1, 0, 0, 0],    # i 1
[0, 0, 1, 0, 0],    # e 2
[0, 0, 0, 1, 0],    # l 3
[0, 0, 0, 0, 1],    # o 4
```

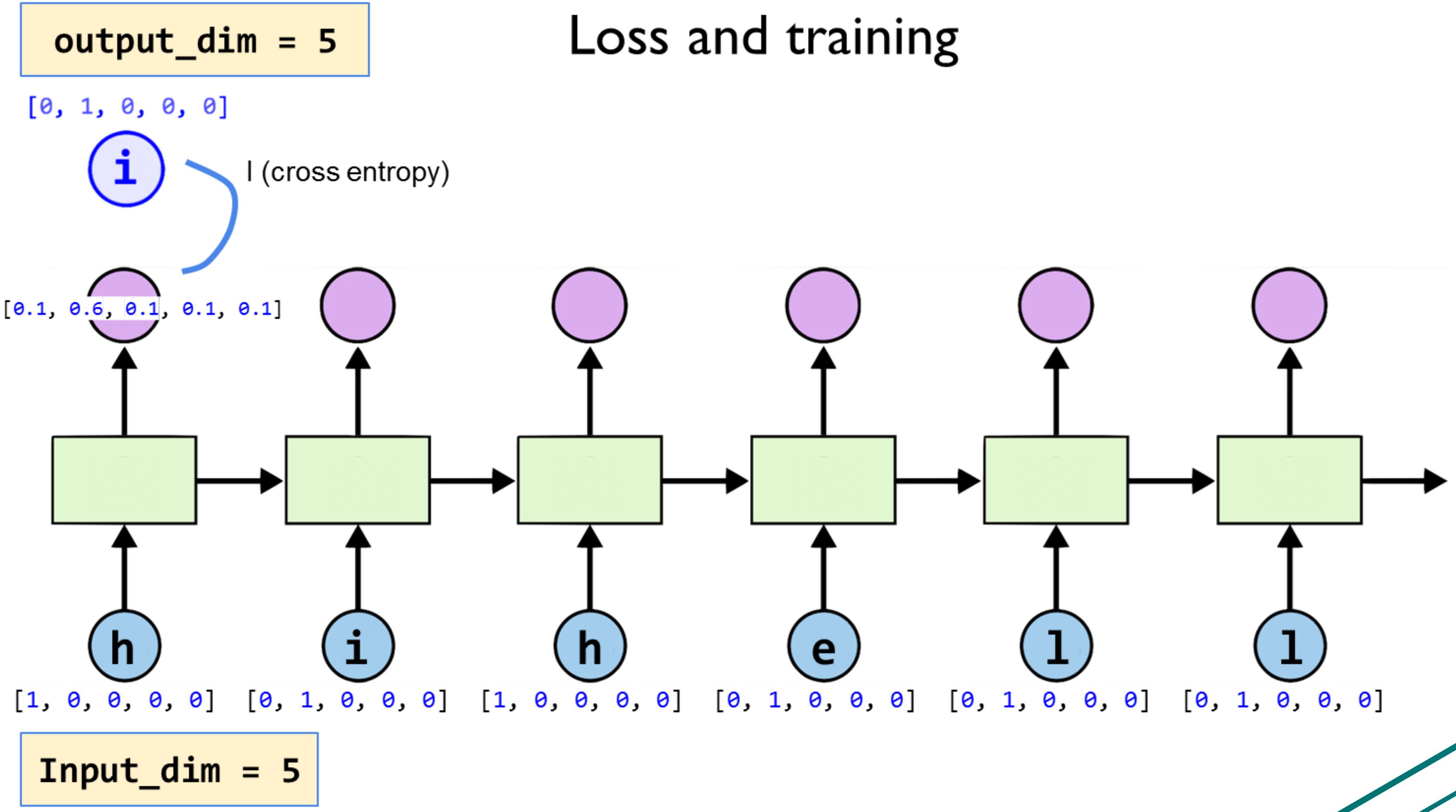
output_dim = 5



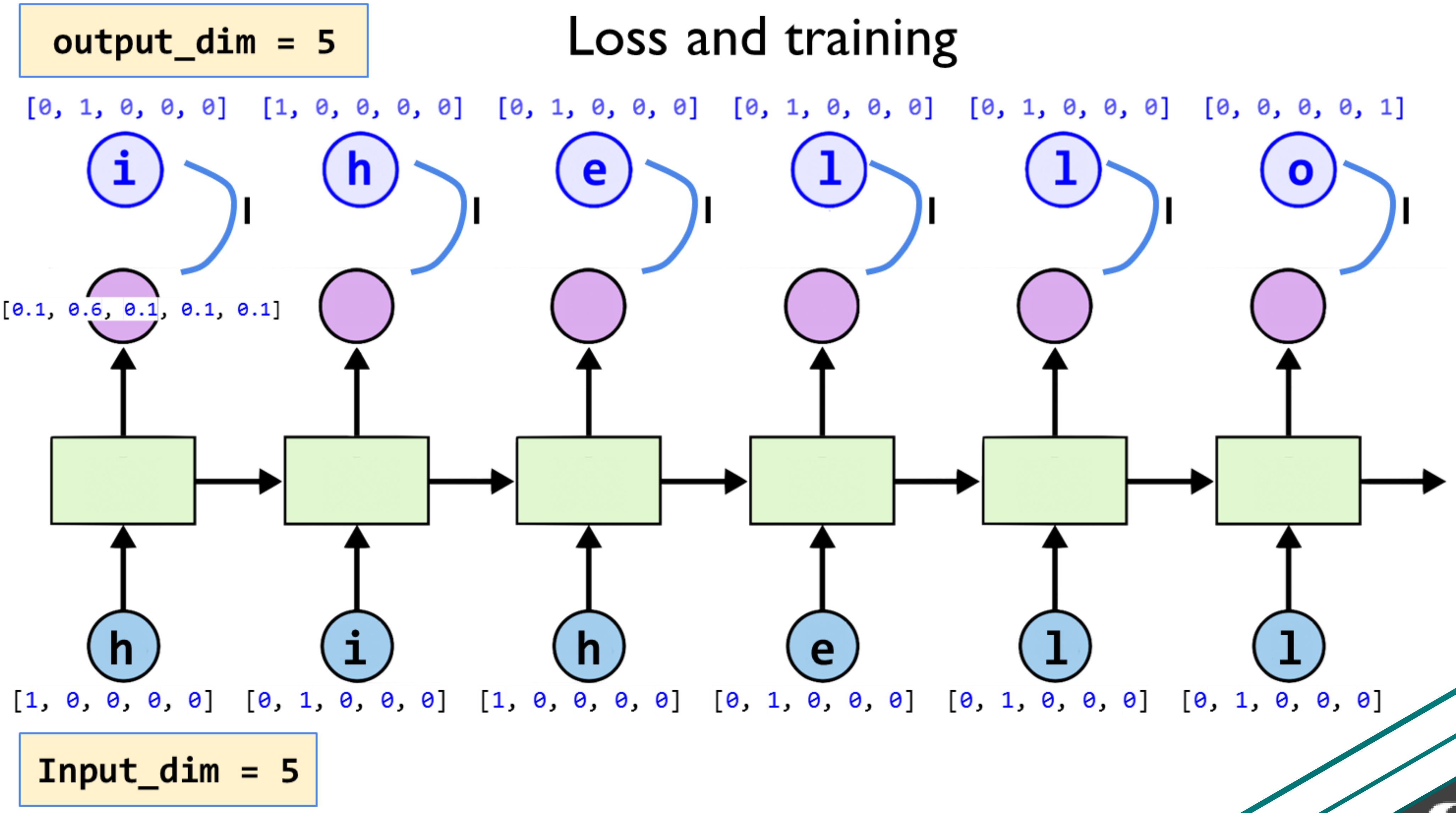
Input_dim = 5

مرکز تحقیقات هوش پارت

Loss and training



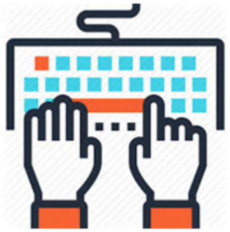
Loss and training



(1)Data preparation 1

```
idx2char = ['h', 'i', 'e', 'l', 'o']

# Teach hihell -> ihello
x_data = [[0, 1, 0, 2, 3, 3]]   # hihell
x_one_hot = [[[1, 0, 0, 0, 0],   # h 0
              [0, 1, 0, 0, 0],   # i 1
              [1, 0, 0, 0, 0],   # h 0
              [0, 0, 1, 0, 0],   # e 2
              [0, 0, 0, 1, 0],   # l 3
              [0, 0, 0, 1, 0]]]  # l 3

y_data = [1, 0, 2, 3, 3, 4]    # ihello

# As we have one batch of samples, we will change them to variables only once
inputs = Variable(torch.Tensor(x_one_hot))
labels = Variable(torch.LongTensor(y_data))
```

(1)Data preparation 2



```
idx2char = ['h', 'i', 'e', 'l', 'o']

# Teach hihell -> ihello
x_data = [0, 1, 0, 2, 3, 3]   # hihell
one_hot_lookup =[[1, 0, 0, 0, 0],  # 0
                 [0, 1, 0, 0, 0],  # 1
                 [0, 0, 1, 0, 0],  # 2
                 [0, 0, 0, 1, 0],  # 3
                 [0, 0, 0, 0, 1]]  # 4

y_data = [1, 0, 2, 3, 3, 4]    # ihello
x_one_hot = [one_hot_lookup[x] for x in x_data]

# As we have one batch of samples, we will change them to variables only once
inputs = Variable(torch.Tensor(x_one_hot))
labels = Variable(torch.LongTensor(y_data))
```

(2) Parameters

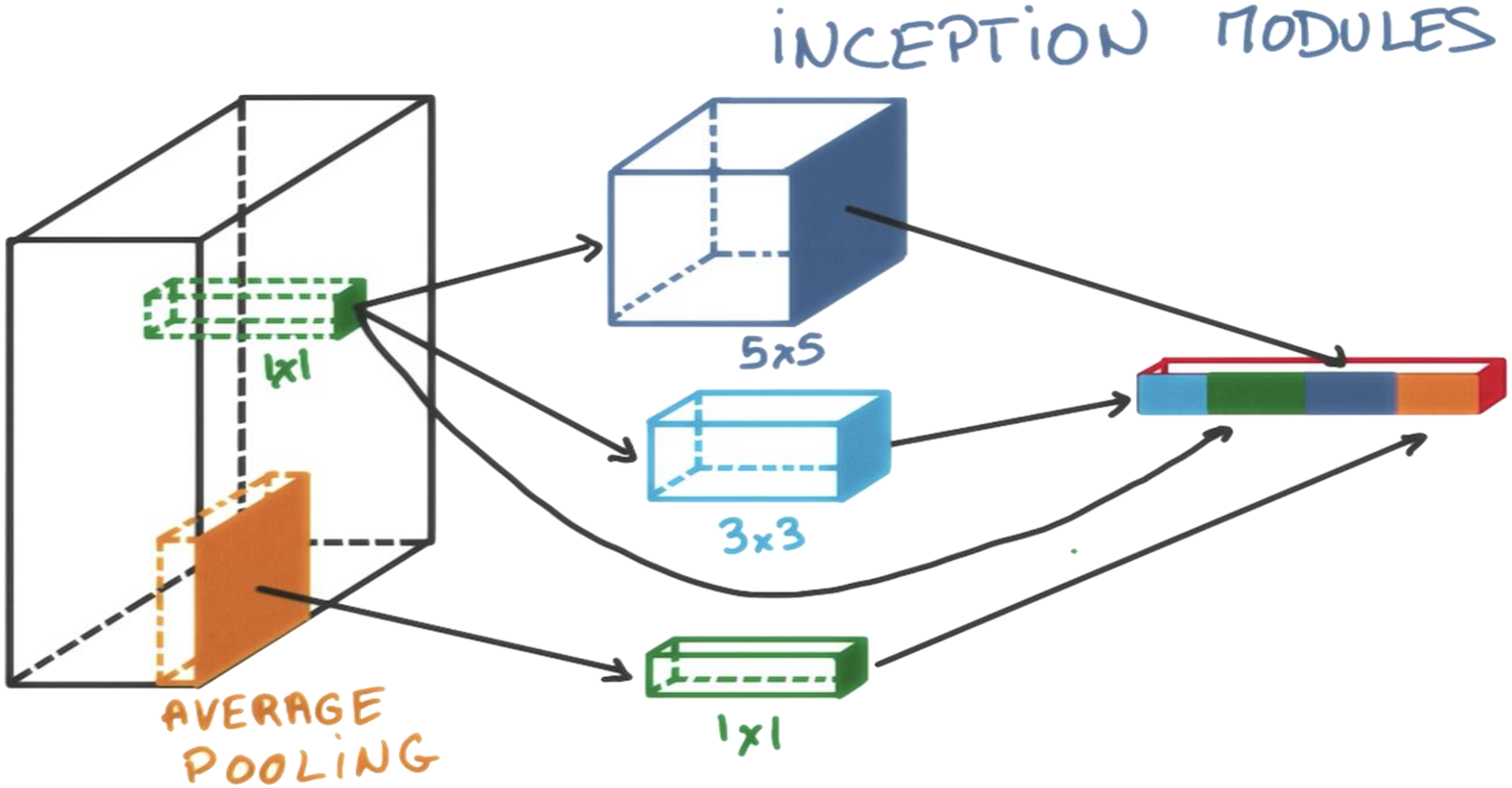
```
num_classes = 5
input_size = 5  # one-hot size
hidden_size = 5  # output from the LSTM. 5 to directly predict one-hot
batch_size = 1   # one sentence
sequence_length = 1  # Let's do one by one
num_layers = 1  # one-layer rnn
```

(3) Our model

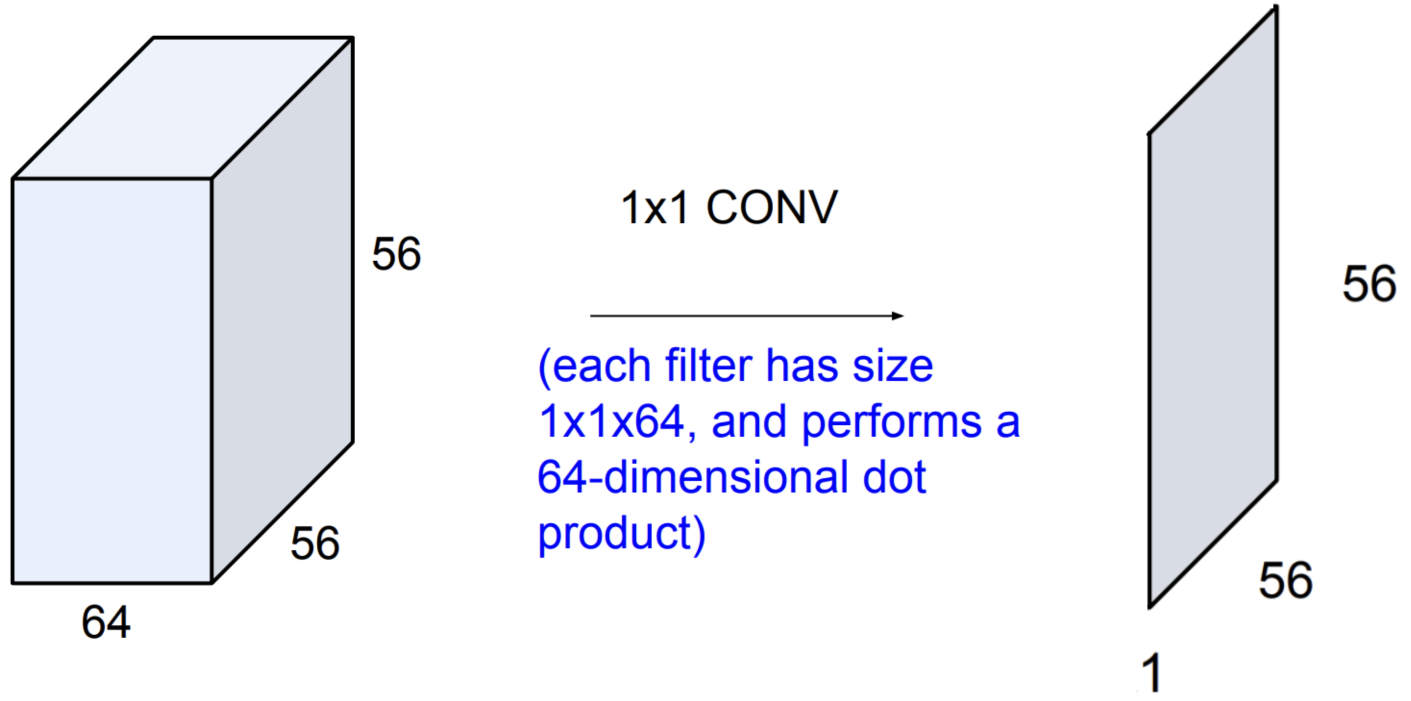
```python
class Model(nn.Module):

    def __init__(self):
        super(Model, self).__init__()
        self.rnn = nn.RNN(input_size=input_size,
                          hidden_size=hidden_size, batch_first=True)

    def forward(self, hidden, x):
        # Reshape input in (batch_size, sequence_length, input_size)
        x = x.view(batch_size, sequence_length, input_size)

        # Propagate input through RNN
        # Input: (batch, seq_len, input_size)
        # hidden: (batch, num_layers * num_directions, hidden_size)
        out, hidden = self.rnn(x, hidden)
        out = out.view(-1, num_classes)
        return hidden, out

    def init_hidden(self):
        # Initialize hidden and cell states
        # (batch, num_layers * num_directions, hidden_size) for batch_first=True
        return Variable(torch.zeros(batch_size, num_layers, hidden_size))
```

```python
num_classes = 5
input_size = 5  # one-hot size
hidden_size = 5  # output from the LSTM.
batch_size = 1   # one sentence
sequence_length = 1
num_layers = 1  # one-layer rnn
```

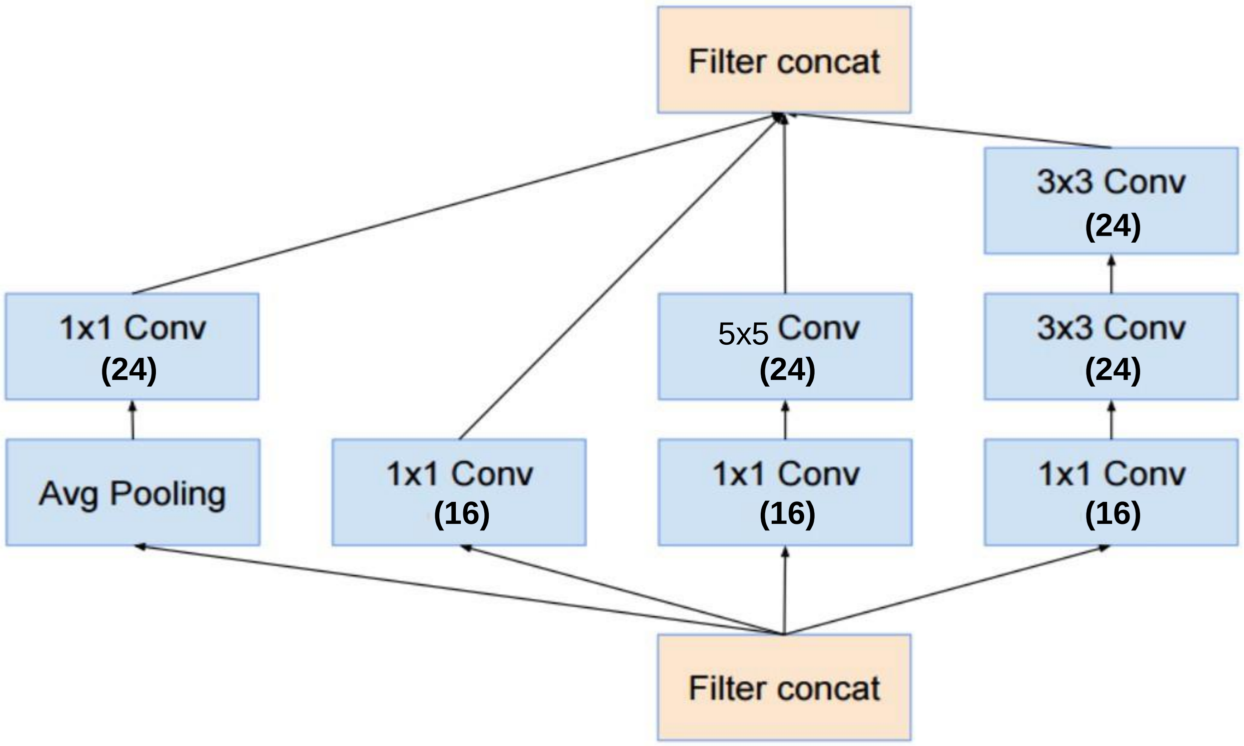
Inception Modules



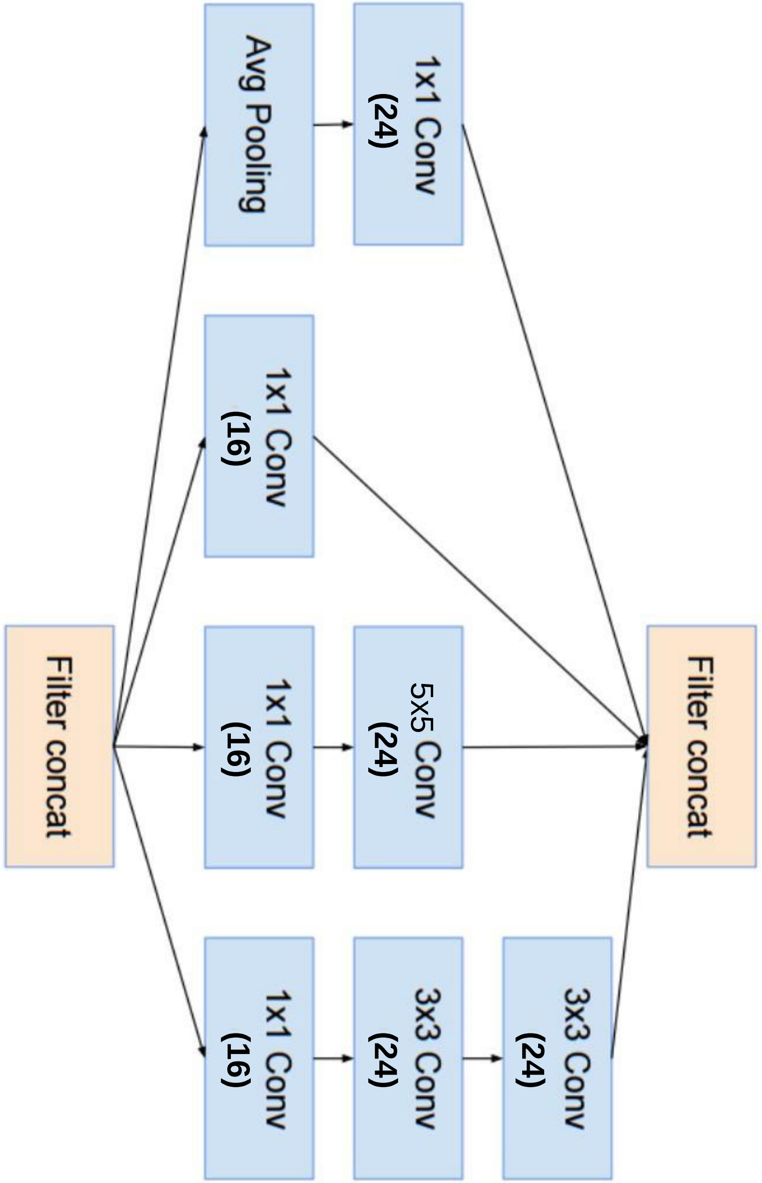
Why 1x1 convolution?

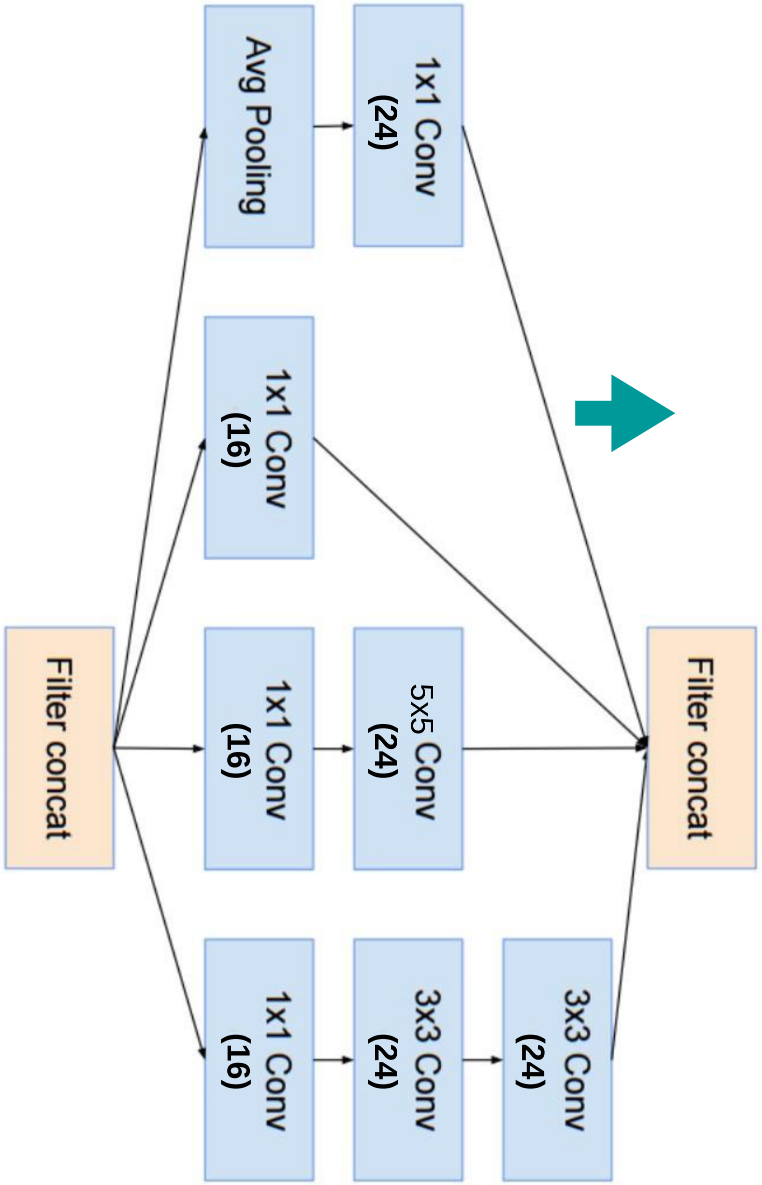


Inception Module



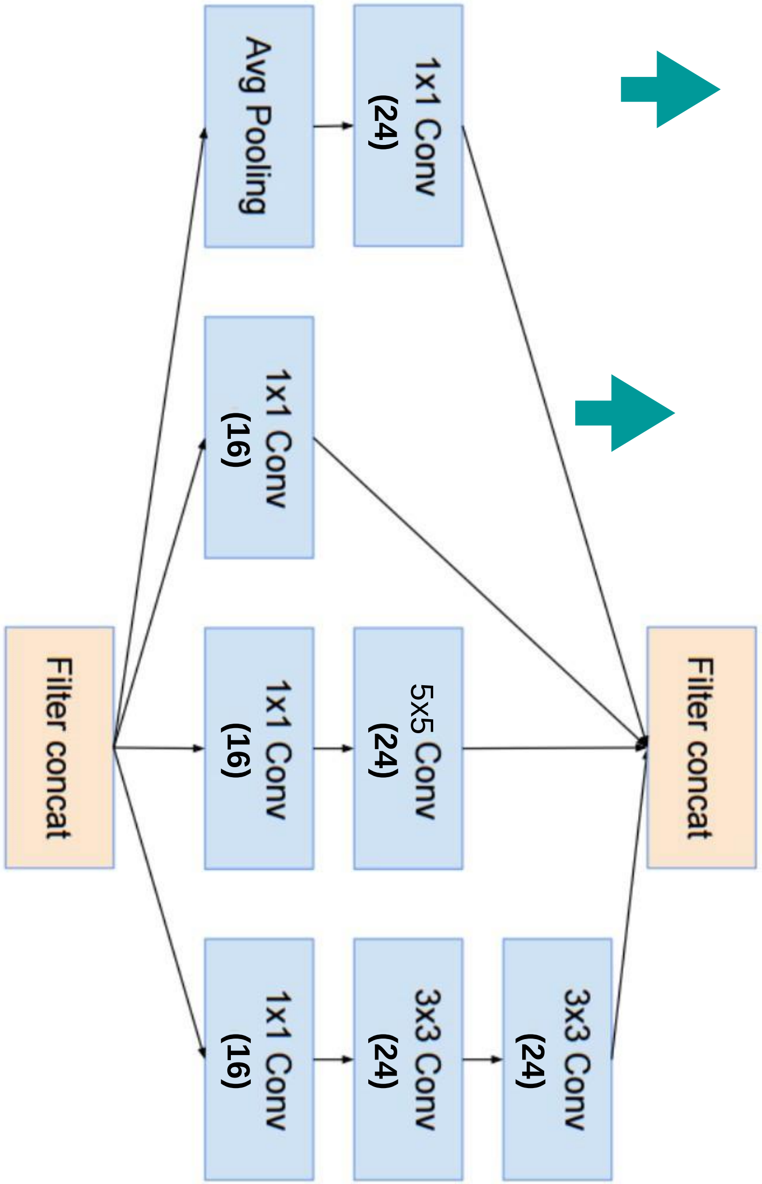
مرکز تحقیقات هوش پارت


Filter concat
1x1 Conv (16)
1x1 Conv (16)
1x1 Conv (16)
Avg Pooling
3x3 Conv (24)
5x5 Conv (24)
1x1 Conv (24)
3x3 Conv (24)
Filter concat



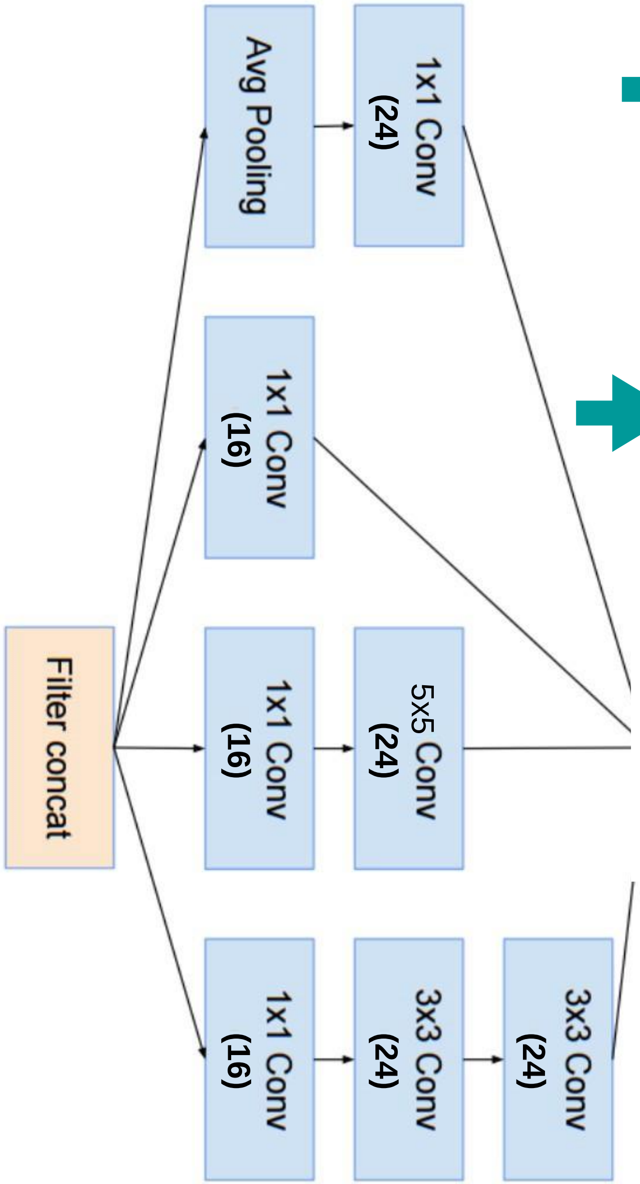

```
self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)

branch1x1 = self.branch1x1(x)
```

مرکز تحقیقات هوش پارت




```
self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
branch_pool = self.branch_pool(branch_pool)
```

```
self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)
branch1x1 = self.branch1x1(x)
```

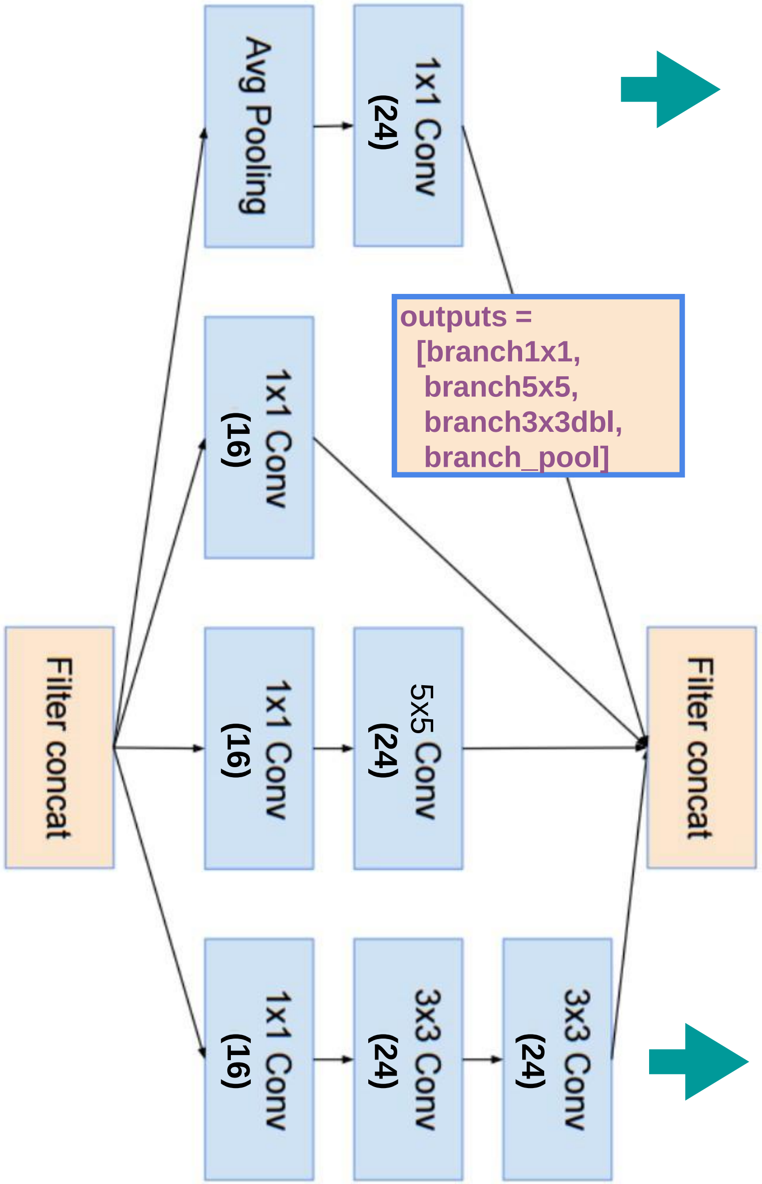



```python
self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
branch_pool = self.branch_pool(branch_pool)
```

```python
self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)
branch1x1 = self.branch1x1(x)
```

```python
self.branch5x5_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
self.branch5x5_2 = nn.Conv2d(16, 24, kernel_size=5, padding=2)

branch5x5 = self.branch5x5_1(x)
branch5x5 = self.branch5x5_2(branch5x5)
```

```python
self.branch3x3dbl_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
self.branch3x3dbl_2 = nn.Conv2d(16, 24, kernel_size=3, padding=1)
self.branch3x3dbl_3 = nn.Conv2d(24, 24, kernel_size=3, padding=1)

branch3x3dbl = self.branch3x3dbl_1(x)
branch3x3dbl = self.branch3x3dbl_2(branch3x3dbl)
branch3x3dbl = self.branch3x3dbl_3(branch3x3dbl)
```




```
self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
branch_pool = self.branch_pool(branch_pool)
```

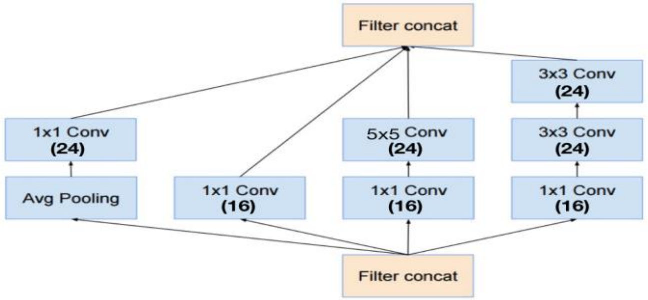
```
self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)

branch1x1 = self.branch1x1(x)
```

```
self.branch5x5_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
self.branch5x5_2 = nn.Conv2d(16, 24, kernel_size=5, padding=2)

branch5x5 = self.branch5x5_1(x)
branch5x5 = self.branch5x5_2(branch5x5)
```

```
self.branch3x3dbl_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
self.branch3x3dbl_2 = nn.Conv2d(16, 24, kernel_size=3, padding=1)
self.branch3x3dbl_3 = nn.Conv2d(24, 24, kernel_size=3, padding=1)

branch3x3dbl = self.branch3x3dbl_1(x)
branch3x3dbl = self.branch3x3dbl_2(branch3x3dbl)
branch3x3dbl = self.branch3x3dbl_3(branch3x3dbl)
```

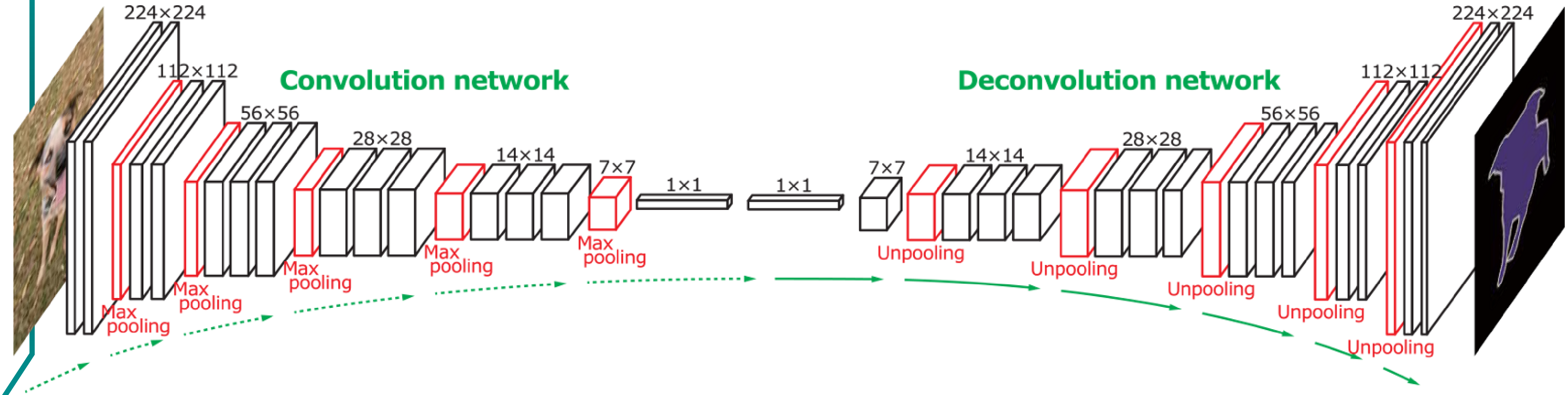
Inception Module

```python
class InceptionA(nn.Module):
    def __init__(self, in_channels):
        super(InceptionA, self).__init__()
        self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)

        self.branch5x5_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
        self.branch5x5_2 = nn.Conv2d(16, 24, kernel_size=5, padding=2)

        self.branch3x3dbl_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
        self.branch3x3dbl_2 = nn.Conv2d(16, 24, kernel_size=3, padding=1)
        self.branch3x3dbl_3 = nn.Conv2d(24, 24, kernel_size=3, padding=1)

        self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

    def forward(self, x):
        branch1x1 = self.branch1x1(x)

        branch5x5 = self.branch5x5_1(x)
        branch5x5 = self.branch5x5_2(branch5x5)

        branch3x3dbl = self.branch3x3dbl_1(x)
        branch3x3dbl = self.branch3x3dbl_2(branch3x3dbl)
        branch3x3dbl = self.branch3x3dbl_3(branch3x3dbl)

        branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
        branch_pool = self.branch_pool(branch_pool)

        outputs = [branch1x1, branch5x5, branch3x3dbl, branch_pool]
        return torch.cat(outputs, 1)
```




```python
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10,
kernel_size=5)
        self.conv2 = nn.Conv2d(88, 20,
kernel_size=5)

        self.incept1 = InceptionA(in_channels=10)
        self.incept2 = InceptionA(in_channels=20)

        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(1408, 10)

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = self.incept1(x)
        x = F.relu(self.mp(self.conv2(x)))
        x = self.incept2(x)
        x = x.view(in_size, -1)  # flatten the
tensor
        x = self.fc(x)
        return F.log_softmax(x)
```

Transpose Convolution



Deconvolution and Unpooling

- Unpooling