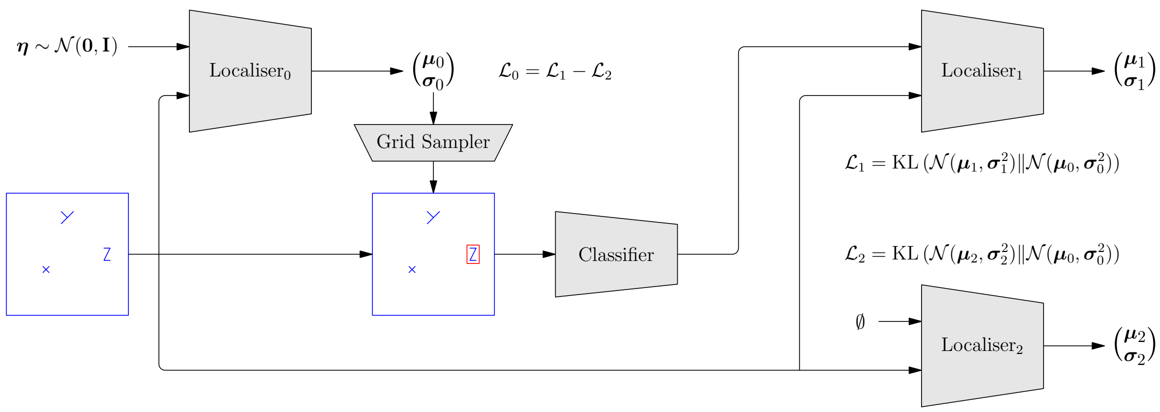
Designing a Network

Adam Prügel-Bennett

January 13, 2021

1 Overall Design

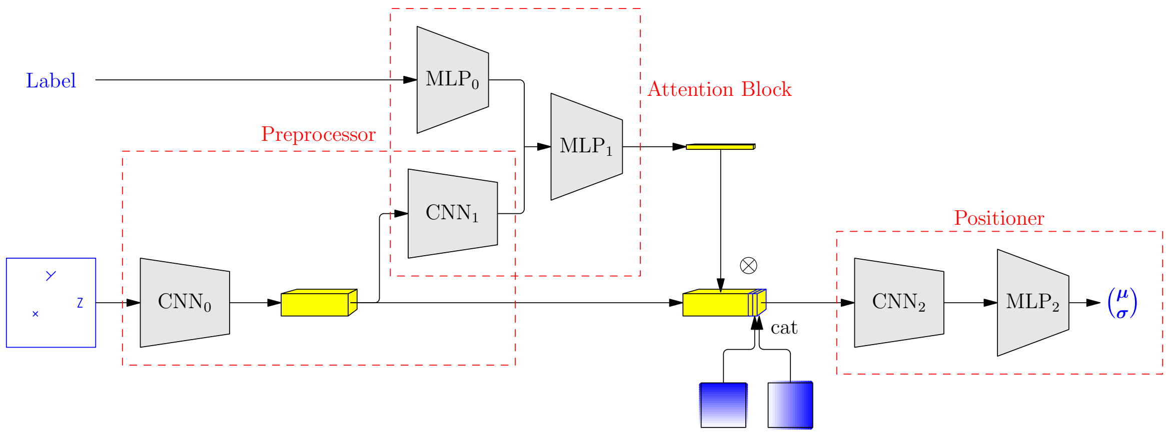


NoP-Learner Architecture

- Localisers differ by attention unit
- Use of `affine_grid` and `grid_sampler` to sample a local part of image
- Converting $(x, y, \sigma_x, \sigma_y)$ to affine parameters $\begin{pmatrix} \sigma_x & 0 & x \\ 0 & \sigma_y & y \end{pmatrix}$

```
import torch
import torch.nn.functional as F
b = 3;                      # batch size
h = w = 4;
c = 3;
images = torch.rand([b,c,h,w])
newHeight = newWidth

t = torch.rand([b,4])  # example output for localiser0
toAffine = torch.tensor([[[0.0,0,1,0],[0,0,0,0],[1,0,0,0]],[[0,0,0,0],[0,0,0,1],[0,1,0,0]]]
a=torch.einsum("xyz,bz->bxy", [toAffine,t])
grid = F.affine_grid(a, [b,c,newHeight,newWidth])
newImages = F.grid_sample(images, grid)
```

2 Localiser



Localiser Architecture

- CNN_0, CNN_1, CNN_2 and MLP_3 are all shared between localisers
- They differ only in the attention block and there only in MLP_1 and MLP_2
- Number of channels output by CNN_0 will depend on complexity of input
  - (c, w, h) = (16,16,16) seems reasonable

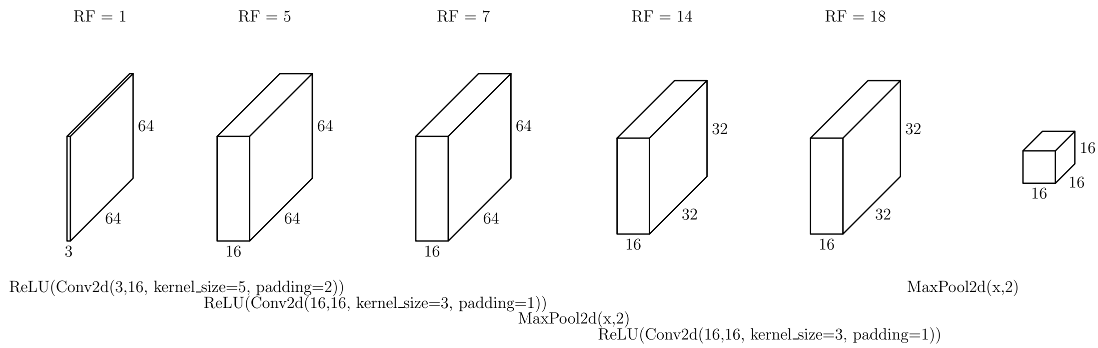
2.1 Preprocessor

_ This is CNN_0 and CNN_1

- Turns input image into 16x16x16 tensor
- This is split into two parts

2.1.1 Preprocessor part 1

- CNN1

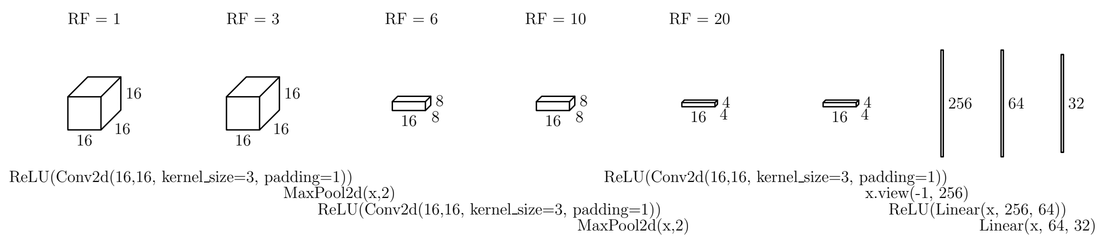



```
# init
        def __init__(self):
                super(Net, self).__init__()
                self.conv1 = nn.Conv2d(3,16, kernel_size=5, padding=2)
                self.conv2 = nn.Conv2d(16,16, kernel_size=3, padding=1)
                self.conv3 = nn.Conv2d(16,16, kernel_size=3, padding=1)

# forward
        def forward(self, x):
                x = F.relu(self.conv1(x))
```

```
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = F.max_pool2d(F.relu(self.conv3(x)), 2)
        return x
```

2.1.2 Preprocessor Part 2

- This is CNN_1



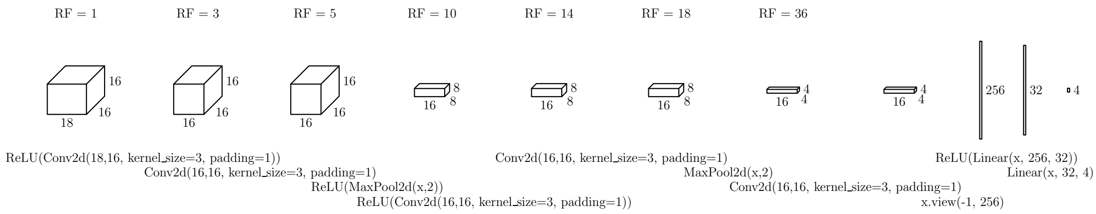

```
# init
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(16,16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16,16, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(16,16, kernel_size=3, padding=1)
        self.fc1 = nn.linear(256, 64),
        self.fc2 = nn.linear(64, 32),

# forward
    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), 2)
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = F.relu(self.conv3(x))
        x = x.view(-1, 256)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

2.2 Positioner

- This is CNN_2 and MLP_3
- Takes output from preprocessor and attention and returns pos=[$\mu_{\text{x},\mu\text{y},\sigma\text{x,y}}$]
- Concatenate two feature maps with x and y position using torch.cat (see CoordConv])

```
import torch
w = 4 # assuming h=w
b = 2
c = 3
ones = torch.ones(w)
seq = torch.linspace(0,1,w)
colCoord = torch.einsum("a,b->ab", [ones,seq]).repeat(b,1,1,1)
rowCoord = torch.einsum("a,b->ab", [seq,ones]).repeat(b,1,1,1)
t = torch.ones([b,c,w,w])
tcat = torch.cat((t,colCoord,rowCoord), dim=1)
print(tcat)
```

- CNN2, MLP2




```
# init
        def __init__(self):
                super(Net, self).__init__()
                self.conv1 = nn.Conv2d(18,16, kernel_size=3, padding=1)
                self.conv2 = nn.Conv2d(16,16, kernel_size=3, padding=1)
                self.conv3 = nn.Conv2d(16,16, kernel_size=3, padding=1)
                self.conv4 = nn.Conv2d(16,16, kernel_size=3, padding=1)
                self.conv5 = nn.Conv2d(16,16, kernel_size=3, padding=1)
                self.fc1 = nn.linear(256, 32),
                self.fc2 = nn.linear(32, 4),

# forward
        def forward(self, x):
                x = F.relu(self.conv1(x))
                x = F.relu(F.max_pool2d(self.conv2(x), 2))
                x = F.relu(self.conv3(x))
                x = F.max_pool2d(self.conv4(x), 2)
                x = self.conv5(x)
                x = x.view(-1, 256)
                x = F.relu(self.fc1(x))
                x = self.fc2(x)
                return x
```

2.3 Attention Module

- With no input (Locaiser$_2$) we don't have MLP$_1$
- Number of outputs = Number of feature sets (channels)
- Multiply channels by output

```
import torch
t = torch.ones([2,3,4,4])
print(t)
att = torch.tensor([[1,2,3],[2,3,4]])
ta = torch.einsum("bcwh,bc->bcwh", [t,att])
print(ta)
```

3 Loss functions

We consider three sets of parameters

1. Localisation parameters, Classifier and Attention1
    - Minimise $\mathcal{L}_1 = \mathrm{KL}(q_0 \| q_1)$
2. Attention2
    - Minimise $\mathcal{L}_2 = \mathrm{KL}(q_0 \| q_2)$

3. Attention0
    - Minimise $\mathcal{L}_1 - \mathcal{L}_2$

3.1 KL-losses

- KL-divergence for general probabilities

$$\mathrm{KL}(q_0 \| q_1) = \int q_0(\boldsymbol{x}) \log\left(\frac{q_0(\boldsymbol{x})}{q_1(\boldsymbol{x})}\right) \mathrm{d}x$$

- Two normals

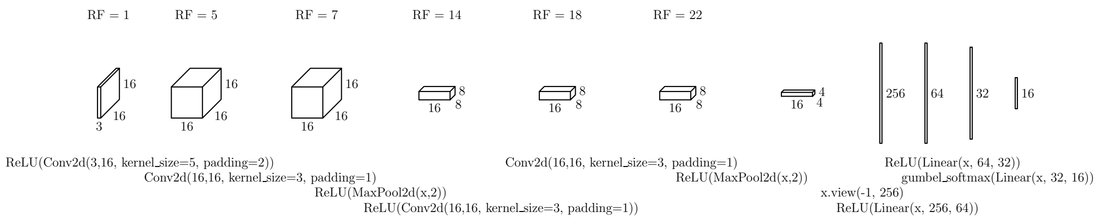
$$\mathrm{KL}(q_0 \| q_1) = \frac{1}{2}\left(\frac{\sigma_0^2}{\sigma_1^2} - 1 - \log\left(\frac{\sigma_0^2}{\sigma_1^2}\right) + \frac{(\mu_0 - \mu_1)^2}{\sigma_1^2}\right)$$

- I prefer to output σ_i as it is dimensionally meaningful. Also I know that $0 < \sigma_i < 1$ so I can put this through a sigmoid

```
kl_loss = 0.5 * torch.sum(torch.exp(z_var) + z_mu**2 - 1. - z_var)
```

4 Classifier

- Input: 16x16 subimage
- The classifier is a small CNN using Gumbel softmax pytorch code pytorch docs
- We can experiment with multiple outputs as an example of disentanglement
- Assuming sub-images of size (3,16,16)




```
import torch
import torch.nn as nn
import torch.nn.functional as F

NoInChannels = 3;

class Classifier(nn.Module):

        def __init__(self):
                super(Classifier, self).__init__()
                self.conv1 = nn.Conv2d(NoInchannels,16, kernel_size=5, padding=2)
                self.conv2 = nn.Conv2d(16,16, kernel_size=3, padding=1)
                self.conv3 = nn.Conv2d(16,16, kernel_size=3, padding=1)
                self.conv4 = nn.Conv2d(16,16, kernel_size=3, padding=1)
                self.fc1 = nn.Linear(256, 64)
                self.fc2 = nn.Linear(64, 32)
                self.fc3 = nn.Linear(32, 16)
```

```
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = F.relu(self.conv3(x))
        x = F.relu(F.max_pool2d(self.conv4(x), 2))
        x = x.view(-1, 256)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.gumbel_softmax(self.fc3(x), hard=True)
        return x

classifier = Classifier();

batchsize = 3

x = torch.rand([batchsize, NoInChannels, 16, 16])

to = classifier.forward(x)
to.shape
```

5 Datasets

- MultiMNist
  - 256x256
- CLEVR
  - 128x128
- Coco