Task 1. Paper Review

Title: Attention Based Glaucoma Detection: A Large-Scale Database and CNN Model
Authors: ***Liu Li, Mai Xu, Xiaofei Wang, Lai Jiang, Hanruo Liu***
Link: pdf
Year: 2019
Code: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2019, pp. 10571-10580

What: The paper propose incorportate Attention maps in CNN architecture in order to get rid of redundancy in medical images related to glaucoma diagnostics and improve general accuracy.
How:
Authors state that approach with localization of ROI using real human(doctors) input wasn't used before. One of the main achievements of the described work is creation of the unique dataset.
Implementation details:

- Attention maps - are 'heatmaps' or areas of the image where usually doctors are fucused when analysing images for glaucoma. These maps are captured with special equipment and unique dataset is used.
- The authors propose architecture called AG-CNN. Since the glaucoma diagnosis is highly related to small ROI regions of the image special CNN architecture is developed. The CNN consists of 3 subnets:
  Attention Prediction subnet, Pathological Area Localization and Glaucoma Classification subnet.
- The main structure of AG-CNN is based on residual networks, in which the basic module is building block. Note that all convolutional layers in AG-CNN are followed by a batch normalization layer and a ReLU layer for increasing the nonlinearity of AG-CNN
- The process of training AG-CNN is in an end-to-end manner with three parts of supervision, i.e., attention prediction loss, pathological area localization loss and glaucoma classification loss.
- *Attention Prediction subnet* - is CNN that is designed to generate attention maps, which are used for pathological area localization and glaucoma detection. Input is for this subnet is image in tensor form (size:224x224x3), the output of this subnet is grey attention map with size of 112x112x1. In AG-CNN, the yielded attention maps are used to weight the input fundus images and the extracted features of the pathological area localization subnet
- *Pathological area localization subnet* - is designed to visualize the CNN features for finding the pathological area. And mainly composed of convlutional layers and fully connected layers and uses image and attention prediction map to produce pathological area and ouputs images of size 112x112x1 which is the input for the Classification Subnet.

- Classification subnet - is usual CNN with classification output, where input images are additionaly weighted with pathological subnet output.

for the full schema see Task 2.

Results:

The proposed approach showed significant perfomance improvement over previous state of the art works on the created and public datasets for glaucoma.

Table 2. Performance of three methods for glaucoma detection over the test set of our LAG database.

Method	Accuracy	Sensitivity	Specificity	AUC	$F_2-score$
Ours	**95.3%**	**95.4%**	**95.2%**	**0.975**	**0.951**
Chen et al.	89.2%	90.6%	88.2%	0.956	0.894
Li et al.	89.7%	91.4%	88.4%	0.960	0.901

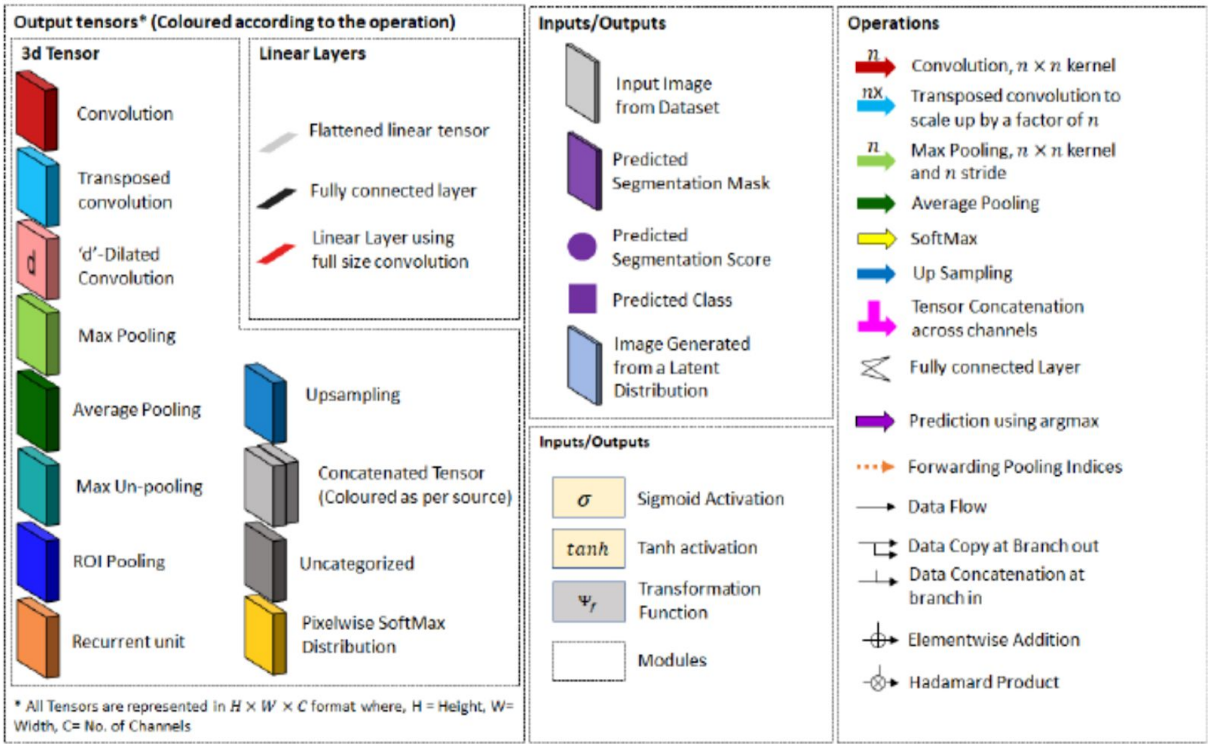
Table 3. Performance of three methods for glaucoma detection over the RIM-ONE database.

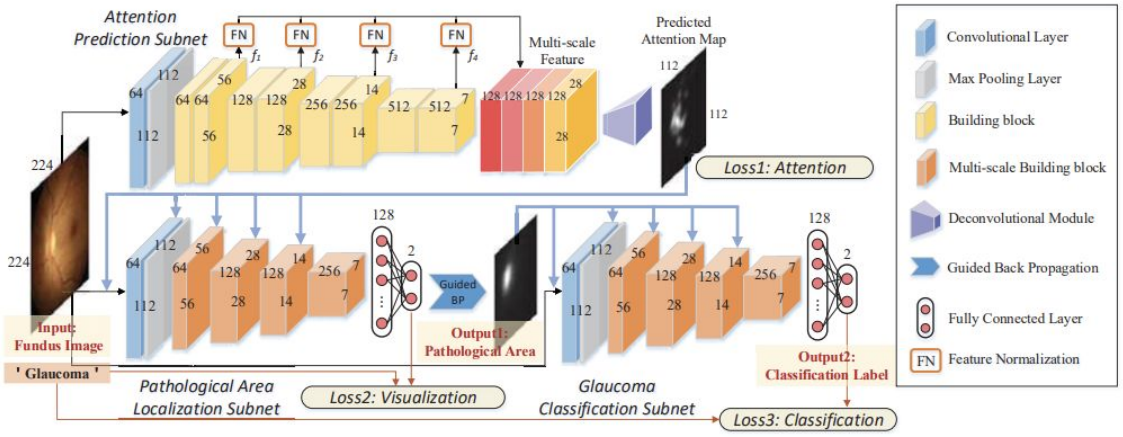
Method	Accuracy	Sensitivity	Specificity	AUC	$F_2-score$
Ours	**85.2%**	**84.8%**	85.5%	**0.916**	**0.837**
Chen et al.	80.0%	69.6%	**87.0%**	0.831	0.711
Li et al.	66.1%	71.7%	62.3%	0.681	0.679

The authors also created llarge-scale attention based glaucoma (LAG) database of images, which is great contribution to the community.

Task 2. AG - CNN architecture visualisation.

Visual vocabulary:





Task 3.

Baseline:

Initial accuracy of the trained net was 64% after 5 epochs. Also the network was noticably overfitted with big difference between loss on test data set and

```
Train Epoch: 1,  train_loss = 3613.7765625 , test_loss = 3681.484130859375
Train Epoch: 2,  train_loss = 3306.783984375 , test_loss = 3438.3671875
Train Epoch: 3,  train_loss = 2608.1283203125 , test_loss = 2842.62451171875
Train Epoch: 4,  train_loss = 2459.875390625 , test_loss = 2778.736328125
Train Epoch: 5,  train_loss = 2091.1275390625 , test_loss = 2547.809326171875
Accuracy of the network on the 10000 test images: 64 %
```

Expriment 1:
Adding batch normalization layer after each convolutional layer.

```
Train Epoch: 1,  train_loss = 3239.6099609375 , test_loss = 3318.913330078125
Train Epoch: 2,  train_loss = 2638.2271484375 , test_loss = 2828.55029296875
Train Epoch: 3,  train_loss = 2481.7923828125 , test_loss = 2790.996337890625
Train Epoch: 4,  train_loss = 2189.129296875 , test_loss = 2583.849853515625
Train Epoch: 5,  train_loss = 1876.170703125 , test_loss = 2390.25048828125
Accuracy of the network on the 10000 test images: 66 %
```

We can see slight improvement over baseline.

Experiment 2:
There is slight overfitting because test_loss is greater than train_loss lets try to add dropout 0.2 dropout for each fully-connected layer.

```
Train Epoch: 1,  train_loss = 2985.8146484375 , test_loss = 3126.818359375
Train Epoch: 2,  train_loss = 2691.6064453125 , test_loss = 2907.408203125
Train Epoch: 3,  train_loss = 2467.22109375 , test_loss = 2712.364501953125
Train Epoch: 4,  train_loss = 2396.9642578125 , test_loss = 2684.420166015625
Train Epoch: 5,  train_loss = 2145.1650390625 , test_loss = 2491.30712890625
Accuracy of the network on the 10000 test images: 64 %
```

Result: learning speed slowered, but overfitting is smaller.

Experiment 3:
Let's try to change optimizer, currently SGD is used.
let's try to change to RMSprop

```
optimizer = torch.optim.RMSprop(net.pararm(), lr=0.001, weight_decay=1e-6, momentum=0.9)
```

Result:

```
Train Epoch: 1,  train_loss = 6022.19375 , test_loss = 6029.7724609375
Train Epoch: 2,  train_loss = 4954.28046875 , test_loss = 4967.04345703125
Train Epoch: 3,  train_loss = 7577.68125 , test_loss = 8055.517578125
Train Epoch: 4,  train_loss = 5113.5078125 , test_loss = 5627.94677734375
Train Epoch: 5,  train_loss = 3783.598046875 , test_loss = 3819.859375
Accuracy of the network on the 10000 test images: 44 %
```

It seams that new optimizer is not improving learning speed, let's revert back to SGD.

Experiment 4:

Currentl we have 3 fully connected layer. I think we can easily get the middle one, because it has no functional meaning in our architecture and just creates additional weights. Let's try the same architecture, but only with 2 FC layers.
Result:

```
Train Epoch: 1,  train_loss = 3200.167578125 , test_loss = 3259.879638671875
Train Epoch: 2,  train_loss = 2896.5861328125 , test_loss = 3040.622802734375
Train Epoch: 3,  train_loss = 2685.3072265625 , test_loss = 2843.890869140625
Train Epoch: 4,  train_loss = 2422.746875 , test_loss = 2669.01904296875
Train Epoch: 5,  train_loss = 2275.5498046875 , test_loss = 2556.960693359375
Accuracy of the network on the 10000 test images: 64 %
```

Remove of the layer had no impact on net performance

Experiment 5:
Now let's try to improve the network acrchitecture by adding more blocks of:

- [conv-relu-pool]XN - [FC]XM

in "Alex-Net" style.
After update our network looks like next:

```
class Net2(nn.Module):
    def __init__(self):
        super(Net2, self).__init__()
        self.conv1 = nn.Conv2d(3, 24, 3)
        self.batch1 = nn.BatchNorm2d(24)
        self.conv2 = nn.Conv2d(24, 48, 3)
        self.batch2 = nn.BatchNorm2d(48)
        self.pool1 = nn.MaxPool2d(2, 2)

        self.conv3 = nn.Conv2d(48, 48, 3)
        self.batch3 = nn.BatchNorm2d(48)
        self.conv4 = nn.Conv2d(48, 48, 3)
        self.batch4 = nn.BatchNorm2d(48)
        self.pool2 = nn.MaxPool2d(2, 2)

        self.conv5 = nn.Conv2d(48, 96, 3)
        self.batch5 = nn.BatchNorm2d(96)
        self.conv6 = nn.Conv2d(96, 96, 3)
        self.batch6 = nn.BatchNorm2d(96)

        self.fc1 = nn.Linear(96, 10)

    def forward(self, x):
        x = self.batch1((F.relu(self.conv1(x))))
        x = self.pool1(self.batch2((F.relu(self.conv2(x)))))

        x = self.batch3((F.relu(self.conv3(x))))
        x = self.pool2(self.batch4((F.relu(self.conv4(x)))))

        x = self.batch5((F.relu(self.conv5(x))))
        x = self.batch6((F.relu(self.conv6(x))))

        x = x.view(-1, 96)
        x = self.fc1(x)
        return x
```

After five epochs results are next:

```
Train Epoch: 1,  train_loss = 3827.5140625 , test_loss = 3842.434326171875
Train Epoch: 2,  train_loss = 3982.29375 , test_loss = 4063.329833984375
Train Epoch: 3,  train_loss = 4516.015625 , test_loss = 4491.2998046875
Train Epoch: 4,  train_loss = 3787.055859375 , test_loss = 3912.683349609375
Train Epoch: 5,  train_loss = 3364.55859375 , test_loss = 3381.496337890625
Accuracy of the network on the 10000 test images: 62 %
```

Training time went significantly up because of the increased count of parameters.

Experiment 6:

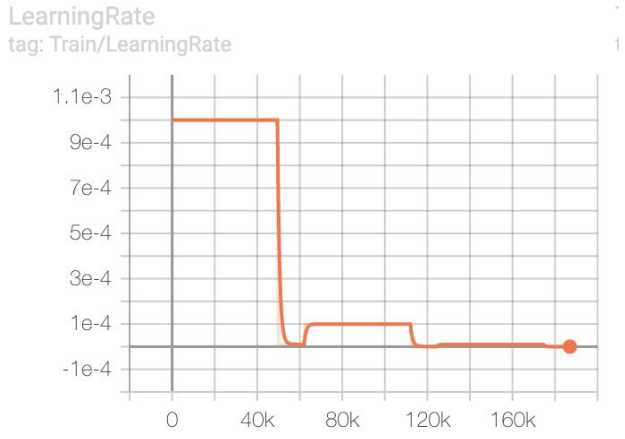
Lets run the 15 epoch training and verify if network accuracy improves over training.

```
Train Epoch: 15,  train_loss = 2192.3111328125 , test_loss = 2368.935791015625
Accuracy of the network on the 10000 test images: 74 %
```

74% is great improvement.


```
Train Epoch: 15,  train_loss = 2192.3111328125 , test_loss = 2368.935791015625
Accuracy of the network on the 10000 test images: 74 %
```


Experiment 7:

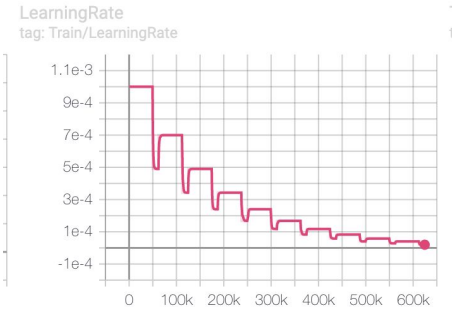


It seams that learning rate is decreasing to fast, and it slows down the training.
Lets implement different learning rate strategy

```
scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                            step_size=3,
                                            gamma=0.5)
```

With this scheduler we learning rate will be decreased by 50% every 3 epoch.

Result:



```
Train Epoch: 50,  train_loss = 956.79345703125 , test_loss = 1704.9609375
Accuracy of the network on the 10000 test images: 76 %
```

We have slight improvemetn we have slight improvement in accuracy, but model is still overfitted, because train_loss is much lover than test_loss.

Experiment 8:

Add data augmentation in order to avoid overfitting.

After change our dataloader looks next:

```
transform_train = transforms.Compose(
    [transforms.RandomRotation(degrees=10),
     transforms.RandomHorizontalFlip(),
     transforms.ColorJitter(hue=.1, saturation=.1),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
     ])
```

I have added random rotation up to 10 degrees, random horizontal flip and color jitter.

Final results:

```
Train Epoch: 50,  train_loss = 1581.9556640625 , test_loss = 1769.9443359375
Accuracy of the network on the 10000 test images: 78 %
```

Next possible improvements:

1. Adding kernel regularization
2. Use pretrained network
3. Use cyclical learning rate