Jörg Horchler Version 1.0, 2019-07-01

1 Definition

1.1 Project Overview

According to World Health Organization[1], breast cancer is the most common cancer among women worldwide. In the United States every 2 minutes a woman is diagnosed with breast cancer and 1 woman will die of breast cancer every 13 minutes on average.

As the lymph nodes in the axilla are most likely the first place breast cancer spreads to these nodes are often removed and examined microscopically. This procedure is demanding and time-consuming. In addition, small metastases are very difficult to detect and sometimes missed.

To assist pathologists the field of digital pathology is developing further. This is mainly supported to advancements in microscopic imaging hardware that allow digitizing glass slides into whole-slide images (WSIs).

Machine learning has shown that image classification and object detection can be automated. Using machine learning for classification of whole-slide images would hold great promise to reduce the workload of pathologists, and reduce the subjectivity in diagnosis.

A modern and proven approach is to use convolutional neural networks (CNNs) trained on patches extracted from these WSIs. Patches are used as WSIs are very large in size and difficult to use as input to CNNs. The aggregate of the predictions of these patches serves as slide-level representation to identify metastases.

Several projects have shown that this approach can outperform pathologists in a variety of tasks [1], [2], [3], [4], [5].

In this project several convolutional neural networks were created on WSI patches to detect tumor metastases.

Two of the networks are trained using “transfer learning” and hence are pre-trained on existing natural images. Only the specific task on detecting tumor metastases was trained on these.

Two other networks were created that are trained from scratch and have no prior knowledge about images at all. One of these two networks use the traditional convolutional layers. The second one uses rotation equivariant layers as proposed by [6] that describes that CNNs are effective on predicting images

> *because all the layers in such a network are translation equivariant: shifting the image and then feeding it through a number of layers is the same as feeding the original image through the same layers and then shifting the resulting feature maps (at least up to edge-effects). In other words, the symmetry (translation) is preserved by each layer, which makes it possible to exploit it not just in the first, but also in higher layers of the network.*

But in contrast to natural images, WSIs exhibit not only translational symmetry but rotation and reflection symmetry as well. Hence a rotation equivariant network should be able to predict tumor patches with a higher accuracy than a standard convolutional network.

1.2 Dataset

There are several datasets of whole-slide images available for example at [7], [8], [9], [10], [11], [12].

[1] http://www.who.int/en/

As WSIs are very large this project used a smaller dataset containing WSI patches available from ([13], [14]). This dataset contains only patches of the original images from [11] and is faster to train.

1.3 Problem Statement

The problem that is to be solved is to detect metastases in hematoxylin and eosin stained images of lymph node sections.

For this the algorithm should be able to discriminate between images with and without metastasis. The result of the task will be the probability of every image to contain metastasis – and therefore whether a patch contains a tumor or not. These will be compared to the ground truth of the dataset.

The main tasks involved are:

- Acquire the datasets.
- Implement a Dataset class for PyTorch to read the datasets.
- Train the two binary classifier using transfer learning. The network used is an inception v3 network available within PyTorch.
- Design and train the two networks that are fully trained. Use a CNN and a G-CNN here.

1.4 Metrics

The models trained were assessed by two metrics:

- Accuracy
- The area under the ROC curve (receiver operating characteristic curve).

Both metrics are a common metric for binary classification tasks.

Accuracy takes into account *true positives* and *true negatives*. It is calculated as

$$\text{accuracy} = \frac{true\ positives + true\ negatives}{dataset\ size}$$

The ROC curve plots the *true positive rate* against the *false positive rate*. These are defined as

$$\text{TPR} = \frac{true\ positives}{true\ positives + false\ negatives}$$

$$\text{FPR} = \frac{false\ positives}{false\ positives + true\ negatives}$$

In sense of tumor detection, the definition of these terms is

		Truth	
		Tumor	Normal
Predicted	Tumor	True Positive	False Positive
	Normal	False Negative	True Negative

To evaluate the **area under the ROC curve** TPR vs. FPR is plotted at different threshold settings, where the threshold is a value between 0 and 1. A lower threshold classifies more items as positive, in other words increases both False Positives and True Positives.

The **AUC** measures the entire two-dimensional area underneath the entire curve from (0,0) to (1,1). AUC ranges in value from 0 to 1 where a model whose predictions are 100% wrong has 0.0; one whose predictions are 100% correct has an AUC of 1.0. Given that a value of 0.5 is the worst because it can be expected that random guessing will produce a ROC curve near the diagonal. A model producing 0.0 can be seen as just interchanging true and false.

2 Analysis

2.1 Data Exploration

The dataset used for this project is based on the dataset of the CAMELYON16 challenge ([11]) . As mentioned in the original challenge the base dataset

> *contains a total of 400 whole-slide images (WSIs) of sentinel lymph node from two independent datasets collected in Radboud University Medical Center (Nijmegen, the*

Netherlands), and the University Medical Center Utrecht (Utrecht, the Netherlands)

As this dataset is very large (around 2 GB each image) a smaller dataset based on these images can be found at [14]. The images obtained from the base dataset are patches (small regions of 96 x 96 Pixels) extracted from the original images. The readme describes the extraction as

> *The slides were acquired and digitized at 2 different centers using a 40x objective (resultant pixel resolution of 0.243 microns). We undersample this at 10x to increase the field of view. We follow the train/test split from the Camelyon16 challenge, and further hold-out 20% of the train WSIs for the validation set. To prevent selecting background patches, slides are converted to HSV, blurred, and patches filtered out if maximum pixel saturation lies below 0.07 (which was validated to not throw out tumor data in the training set). The patch-based dataset is sampled by iteratively choosing a WSI and selecting a positive or negative patch with probability p. Patches are rejected following a stochastic hard-negative mining scheme with a small CNN, and p is adjusted to retain a balance close to 50/50.*

This dataset contains CSV files describing what regions the patches are extracted from. This metadata contains the fields:

- **coord_y**: y-coordinate of the top left pixel of the patch (int64)
- **coord_x**: x-coordinate of the top left pixel of the patch (int64)
- **tumor_patch**: Whether the patch contains at least one pixel of tumor tissue. (bool)
- **center_tumor_patch**: Whether the center 32x32px region contains tumor tissue. (bool)
- **wsi**: The name of the original WSI that was used to extract the patch. This name is encoded as *camelyon16_<split-name>_<label-name>_<wsi-number>*. For example, *camelyon16_train_tumor_090* is training WSI 90 from the original dataset and contains a tumor tissue. (str)

Some example rows are:

coord_y	coord_x	tumor_patch	center_tumor_patch	wsi
63104	43648	False	False	camelyon16_train_tumor_104
148544	74048	True	True	camelyon16_train_tumor_003
64192	78912	True	True	camelyon16_train_tumor_089

Figure 1: CSV example rows

The base dataset is split in train and test images. PCam has done the same split but excluded another 20% of the training set as validation set.

For comparison the original dataset contains

- a total of 400 WSIs split by
  - 270 training images (67.5 %)
  - 130 test images (32.5 %)
  - no validation set.

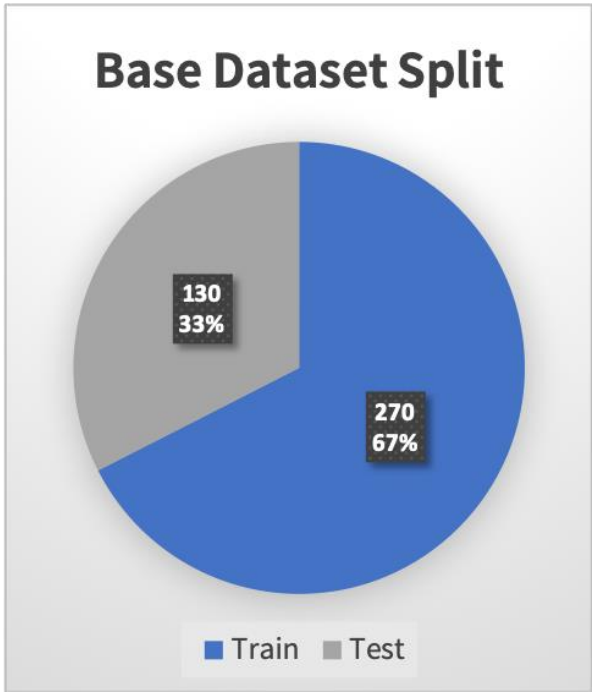

Figure 2: Split of Base Dataset

Analysing the CSV files of PCam showed that it uses only **399** images. One is missing.

This is intended. According to the readme [2]of the original dataset one WSI (test_049) is a duplicate.

The PCam dataset was split according to the numbers of the base dataset seen above. It contains patches from:

- a total of 399 WSIs split into
  - 216 training images (54.14%)
  - 129 test images (32.33%)
  - 54 validation images (3.53%)

As seen the one test image is missing.

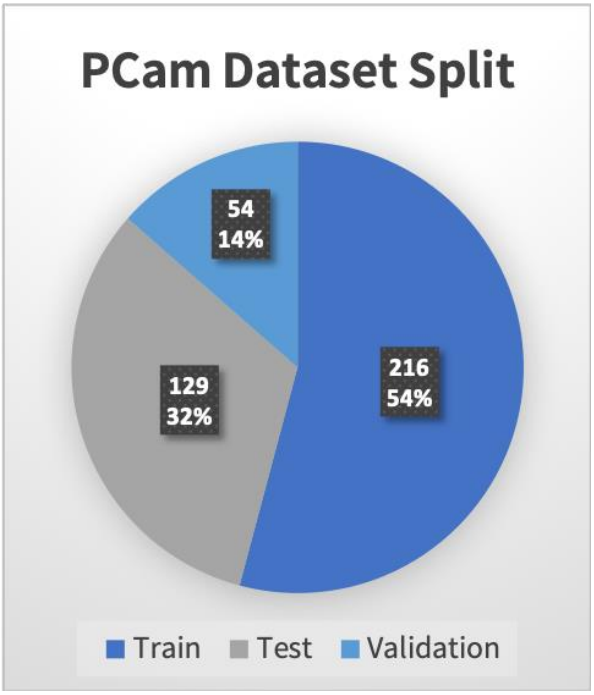

Figure 3: Split of PCam Dataset

The base dataset contains

- 160 images without tumor and
- 110 images with tumor

in the labeled training dataset. The test set contains

- 80 images without tumor and
- 49 images with tumor.

Hence the distribution of both sets is around 40% images labeled with tumor in both splits.

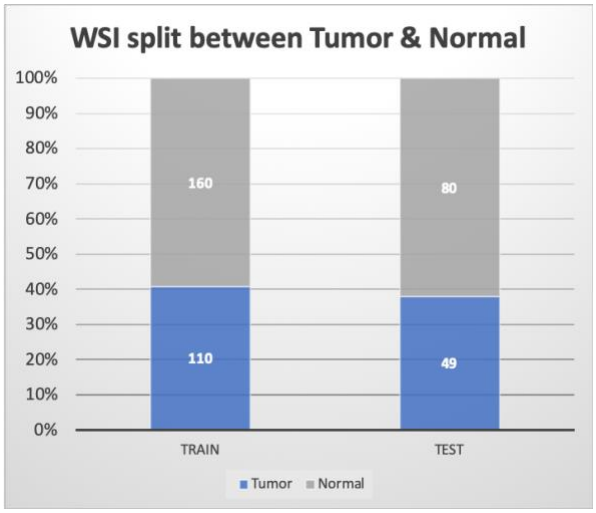

Figure 4: Base dataset split by type

The numbers of the csv files in PCam show that it contains 327.680 images.

It contains tumor patches in

- Train: 131.745
- Validate: 16.458
- Test: 16.474

According to that the normal patches are split with the same schema:

- Train: 130.399
- Validate: 16.310
- Test: 16.294

These numbers show that PCam uses a slightly higher percentage of tumor patches and splits the them by 50% in all splits.

[2] https://camelyon17.grand-challenge.org/media/CAMELYON17/public_html/camelyon16_readme.md

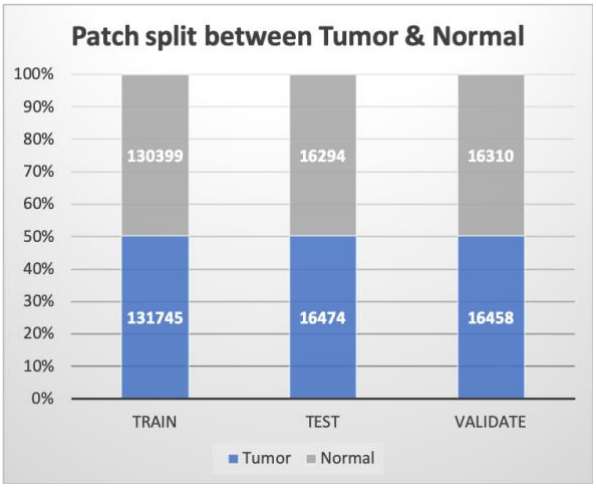

Figure 5: Split of PCam patches by type and set

Rotating the axis in these plots show that over all images the split of tumor and normal images is again higher with the PCam dataset. In the original dataset around 70% of tumor and normal images are contained in the training set and 30% in the test set (train contains 270 images in total, test contains 129 images in total).

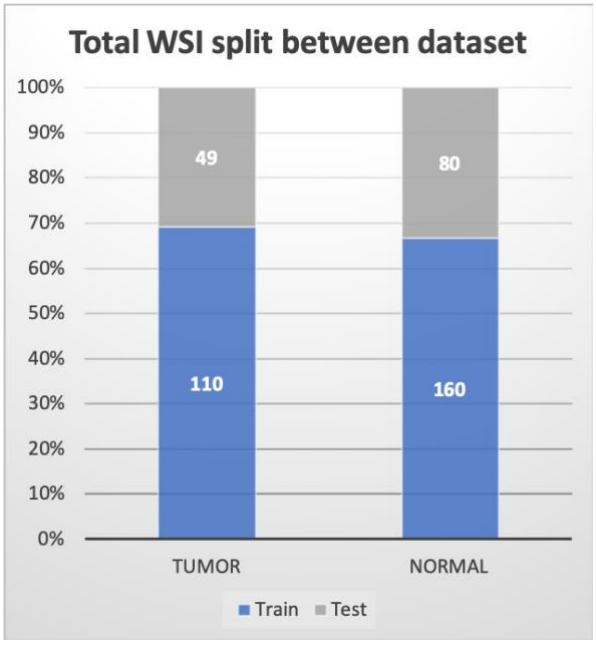

Figure 6: Total split of WSIs between type

The PCam datasets splits these by 80% / 10% / 10%.

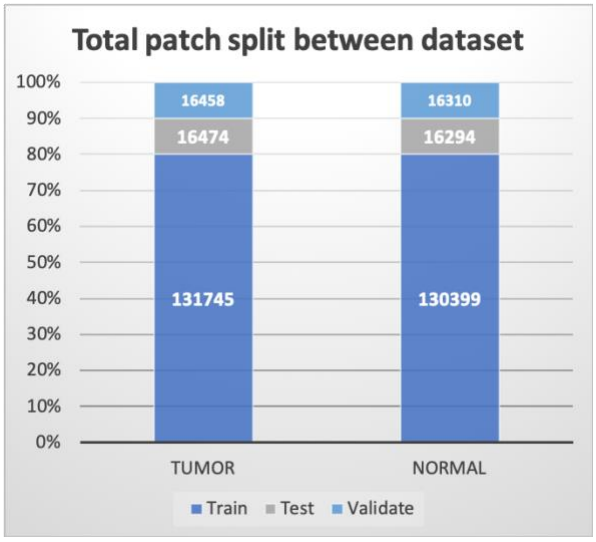

Figure 7: Total split of patches

These numbers are just informative. The models are trained without any modifications of these splits.

Finally, the next figure shows four randomly selected images of the dataset.

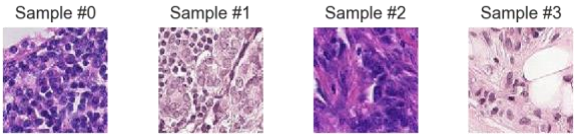

Figure 8: random samples from PCam dataset

2.2 Algorithms and Techniques

The algorithm used is a ***Group Equivariant Convolutional Net*** as proposed in [6]. This is an extension to a ***Convolutional Neural Network*** which is state of the art algorithm for most image processing tasks, including classification.

The framework used to solve the problem is PyTorch which includes the functionality needed to implement neural networks. In addition, an implementation of G-CNNs is available at [15].

2.2.1 Convolutional Neural Network

A Convolutional Neural Network (CNN) is a specialized neural network that is built for images. This is because images share some important properties:

- They are stored as multi-dimensional arrays. For example, '96 x 96 x 3' describes the images in PCam as 96 x 96 pixels for width and height and 3 channels as depth. The three channels describe a value for each color of the image.
- Images feature axis for which ordering matters. For example, width and height as given above.
- The 'channel' axis is used to access different views of the image. These are red, green and blue as mentioned above.

A *normal* neural network does not exploit these properties. It just receives an input matrix and multiplies that with another matrix (the weights) and usually a bias vector is added to the weights too. Hence all axis are treated the same way and topological information is not taken into account.

To overcome these limitations convolutions can be used. These transformations preserve the notation of ordering. In addition, it is sparse because only a few input units contribute to a given output unit. And they can reuse parameters as the same weights are applied to multiple locations in the input.

The convolutions work by sliding a ***filter*** (that is a matrix holding weights) over the image and elementwise multiplying this filter with only the region under the filter, summing it up, and then offsetting the result by the bias. A nice animation of that can be seen at [16].

One can visualize the operation using this image:

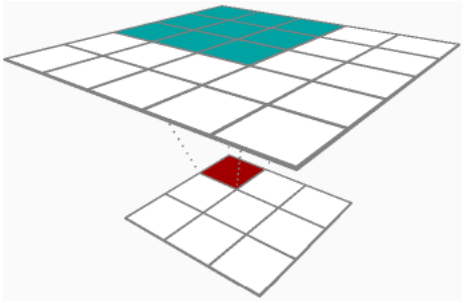

Figure 9: classic 2D convolution[3].

Processing images using convolutions stacks lots of these filters into a larger network in different layers. That way the different layers learn different features of the input image, for example edges, curves and so on.

The output is then an assigned probability for each class.

2.2.2 Group Equivariant CNN

As mentioned in [6] a CNN is equivariant under translation, but not under rotation. That means that a detected feature like a horizontal bar is different from a feature of the same bar if it is rotated. Hence if we rotate an image the resulting feature map is different from the original image. But an hour glass is still an hour glass – even under rotation.

To overcome that a Group Equivariant CNN uses group theory to detect features in an image. In contrast to a CNN it does not use rotated input but rotates the filter used and combines the features extracted to one feature map. That way it can detect the same feature under rotation or other group functions like vertical reflections. A visualisation of rotation is:

[3] http://scyfer.nl/2016/12/13/data-efficient-deep-learning-with-g-cnns/

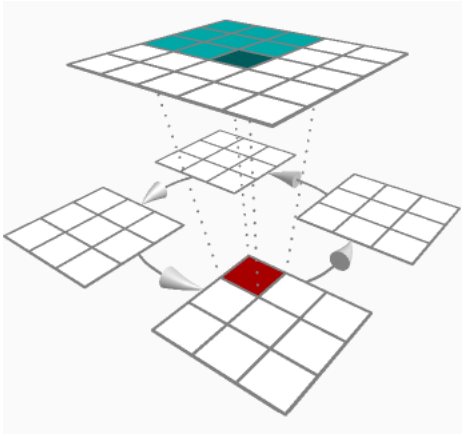

Figure 10: Group Equivariant Convolution[4]

Converting from a CNN to a G-CNN is done by replacing all convolution layers by group equivariant versions.

Implementations can be found at [15], [17] and [18].

2.2.3 Hyperparameters

The hyperparameters to tune a G-CNN are the same as for a CNN:

- Training Parameters
  - Training length (number of epochs)
  - Batch size (how many images to look at once during a single training step; this helps the optimizer updating the gradients)
  - Optimizer type (the algorithm to use for learning)
  - Learning rate (how fast to learn)
  - Weight decay (that prevents the model being dominated by a few neurons having a large weight)
  - Momentum (takes the previous learning step into account when calculating the next one).
- Network Parameters
  - Number of Layers
  - Type of Layers
  - Layer parameters (for example the filter size)
- Data Preprocessing
  - Image augmentations

2.3 Benchmark

Two benchmarks were used to compare the performance of the implemented solution:

- The PCam dataset reports a benchmark achieved using a modified DenseNet at [14]. The score reported was
  - Accuracy: 89.8
  - AUC: 0.96
- In addition, two different Inception v3 networks were trained during the project using transfer learning. The networks were used as fixed feature extractor and were not fully fine tuned as training time was limited. The performance achieved was:
  - Net 1 Accuracy: 81.49
  - Net 1 AUC: 0.87
  - Net 2 Accuracy: 78.12
  - Net 2 AUC: 0.85

Both networks are implemented in the files included in this project:

- cnn/__init__.py
- cnn/helpers.py
- cnn/parameters.py
- utils/ __init__.py
- utils/data.py
- train_inception.py
- test_inception.py

Details will be provided later in this paper.

3 Methodology

3.1 Data Preprocessing

The PCam dataset already has done a lot of work to preprocess the original dataset. All images are stored as arrays in HDF5 files and can be used without conversion. The labels are provided as vectors in HDF5 files too.

[4] http://scyfer.nl/2016/12/13/data-efficient-deep-learning-with-g-cnns/

To use the data and to feed it into the network only a few steps were done:

- As PyTorch can not read HDF5 files directly an own class was implemented. This class is then used in PyTorch's dataloaders.
- As part of the training the image data was converted to PyTorch Tensors using an existing function in PyTorch. This is done on every access of the data as the original HDF5 files were not modified.
- In addition, the images were normalized by subtracting the mean value of the dataset from the pixel values. For the pre-trained networks, the known mean value (and standard deviation) of ImageNet was used. The created model in the project was normalized by first calculating mean and standard deviation over the whole training dataset. This is shown in the Jupyter-Notebook included in the project.

In addition to these steps some image augmentation steps were tried during training. This will be described in a later section.

3.2 Implementation

The implementation was done in several steps:

- To read in the dataset two classes were created:
  - A Dataset class PCamDataset used by PyTorch's dataloader to retrieve mini-batches of the datasets.
  - A HDF5Matrix class that is used to open and read from the dataset downloaded.
- The first neural network was created and trained using existing weights trained on ImageNet.
- Based on the result of this first network a second one was trained by just changing some parameters.
- Finally, a CNN and a G-CNN were created. The CNN was used as a second baseline to see results of a fully trained network.

3.2.1 PyTorch Dataset Class

For access to the dataset a class was created that can access HDF5 files. The class was modeled using the class HDF5Matrix provided by Keras[5]. It uses the h5py module provided by pip.

In addition, the class PCamDataset was created to store a reference to two HDF5 files. This design was chosen to bundle images and labels in one class. That way during training, validation and testing one can use the code

```
(1)  for images, labels in dataloader:
```

to access images and labels in one step.

During the tests it was observed that the standard HDF5 library is not thread safe. Hence it was not possible to open a file in the __init__ section of the class and hold it open until finished when used more than one worker. To overcome that limitation the workaround is to open and close the file on every access in all functions of the HDF5Matrix class. This does not limit the use of the class in the current project but might be in issue in larger projects.

Code of both classes can be found in the file utils/data.py included in the project. To use these two classes the function get_matrixes was created that returns two references to HDF5Matrixes. This design was chosen because the source file names differ only on the phase and contain the string 'train', 'valid' and 'test'. Using this function, the two

[5] https://keras.io/io_utils/

belonging classes for one phase can be access in PyTorch using the code

```
(1)  train_x, train_y = utils.get_matrixes(phase='train',
     transform=data_transforms['train'])
(2)  train_dataset = utils.PCamDataset(train_x, train_y)
(3)  train_loader = DataLoader(train_dataset)
```

3.2.2 Inception v3 Training 1

As baseline the first CNN was created. In the first tests it was seen that finetuning a full network would take 1 hour per epoch on Google Cloud using a Tesla T4 GPU. As this was considered as to much the pretrained network was used as fixed feature extractor only. This reduced training and validation of one epoch to about 20 minutes.

Examples of transfer learning are taken from [19] and [20].

The training and validation code is stored in the files cnn/helpers.py and cnn/parameters.py which are used in file train_inception.py. High level description of the code is:

- Define data transforms.
- Load datasets and create dataloaders.
- Instantiate the model. That includes:
    - Loading the model and the pretrained parameters.
    - Freezing all layers.
    - Replacing the last layers with fully connected layers.
- Define the loss function.
- Define the optimizer.
- Define the scheduler to change the learning rate.
- Train and validate once per epoch.
- Save the best model to file.

3.2.2.1 Data Transforms

This model used no image augmentation. The images were only

- resized to the dimensions expected by Inception (299 x 299 Pixels),
- converted to a PyTorch Tensor,
- channelwise normalized using the mean and standard deviation pre-computed for the ImageNet dataset.

3.2.2.2 Datasets and Dataloaders

These were used as described above using the PCamDataset class.

3.2.2.3 Model creation

PyTorch includes many pretrained models in the package torchvision.models. The model was created using

```
(1)  model = torchvision.models.inception_v3(pretrained=True)
```

Freezing the layers can be done in PyTorch by setting requires_grad = False for all parameters of the model. This ensures that the weights are not updated during backward propagation.

Inception v3 contains an auxiliary branch by default. This is used here to. Due to that two fully connected layers were replaced in the model using

```
(1)  num_features = model.AuxLogits.fc.in_features
(2)  model.AuxLogits.fc = nn.Linear(num_features, num_classes)
(3)  num_features = model.fc.in_features
(4)  model.fc = nn.Linear(num_features, num_classes)
```

Lines 1 and two replaces the prediction layer in the auxiliary branch, 3 and 4 in the main branch. num_classes was set to 2 because the task is a binary classification task.

3.2.2.4 Loss Function

The loss function used during all runs of this project was CrossEntropyLoss. This combines LogSoftmax and NLLLoss in one single class.

This loss is often used in classification tasks having n classes – in this case n = 2.

3.2.2.5 Optimizer

The optimizer chosen was RMSprop with momentum and weight decay. These were chosen because RMSprop with momentum is as fast as the Adam optimizer. Weight decay was used for regularization to prevent biasing the network by some more important neurons having large weights.

The parameters used were

Learning Rate	0.1
Momentum	0.9

Epsilon	1.0
Weight Decay	0.9

Learning rate was set higher than usual (most often we see rates of 0.001 or smaller) because [21] and [22] show that a higher learning rate that is reduced during training can lead to a faster convergence.

Momentum and weight decay as of 0.9 is often used with RMSprop and was just chosen for this reason.

Epsilon was set to 1.0 because in the original paper of Inception it was found out that this was needed[6] for training with async SGD.

3.2.2.6 Learning Rate Scheduler

To avoid training to be stuck in a local minima a scheduler was used to update the learning rate. During the training of the inception network StepLR was used to update the learning rate every 4 epochs by a factor of 0.16.

The factor was just chosen as between 0.1 and 0.2. Most tutorials seen use 0.1.

3.2.2.7 Train and validate

As mentioned above one epoch of training and validation took about 20 minutes. Therefore, both inception models were trained for only 24 epochs.

As PCam contains train and validation data the network was trained and validated on every epoch. This was done using the helper function cnn.helpers.train_and_validate in cnn/helpers.py.

Basically, this function is taken from two PyTorch tutorials at [19] and [20].

A training epoch in PyTorch is done using

```
(1)  model.train()
(2)  for images, labels in trainloader:
(3)    optimizer.zero_grad()
(4)    with torch.set_grad_enabled(True):
(5)      y_preds = model(images)
(6)      loss = loss_fn(y_preds, labels)
(7)      loss.backward()
(8)      optimizer.step()
```

The first line sets the model in training mode. This changes the behavior of some modules, for example dropout only randomly zeroes out some of the elements of the input tensor during training but not during validation.

Line 3 is used to zero all of the gradients for the learnable weights the optimizer will update. This is done because by default, gradients are accumulated in buffers whenever backward() is called. An explanation for the mechanism behind backward() can be seen at [23].

Line 4 enables gradient calculation that builds the backward graph.

Line 5 sends the input data to the forward pass of the network.

Line 6 calculates the loss.

Line 7 does the backward call that calculates the gradients of the tensors w.r.t. graph leaves (see [23]).

Line 8 finally makes the optimizer updates the weights in the network.

In contrast validation is done using

```
(1)  model.eval()
(2)  with torch.no_grad():
(3)    for images, labels in validloader:
(4)      y_preds = model(images)
(5)      loss = loss_fn(y_preds, labels)
```

In this case the model is set to eval mode first as seen in line 1. As mentioned that will change how some modules behave.

Line 2 disables gradient calculation as this is not needed in validation.

Then only the loss is calculated in the next lines. This can be used to calculate accuracy and other statistics.

3.2.3 Inception v3 Training 2

Based on the validation results of the first inception network a second one was created

[6] *https://github.com/ibab/tensorflow-wavenet/issues/149*

that was trained and validated the same way as the first one but with following changes:

- The input images were augmented.
- Instead of RMSprop optimizer SGD was used with Nesterov momentum.

It was expected that both changes would raise accuracy and finally the AUC score.

3.2.3.1 Augmentation

For this network the images were augmented with:

- Random rotations between -45 and +45 degrees.
- With a probability of 0.3 they were randomly flipped horizontally and vertically.
- With a probability of 0.3 they were randomly shifted horizontally and vertically as well as sheared in a range of -45 to +45 degrees.
- With a probability of 0.3 the brightness, contrast, saturation and hue were changed.

3.2.4 Results (Accuracy and Loss)

At the end of train_inception.py the history of loss and accuracy of training and validation was written to csv files. They were merged and analyzed using pandas and the results plotted using matplotlib and seaborn.

For surprise the first model was more accurate than the second. This might be due to two reasons:

- Augmentation of images works like regularization. Hence this might affect learning of the network.
- SGD optimizer using Nesterov momentum is worse compared to RMSprop.

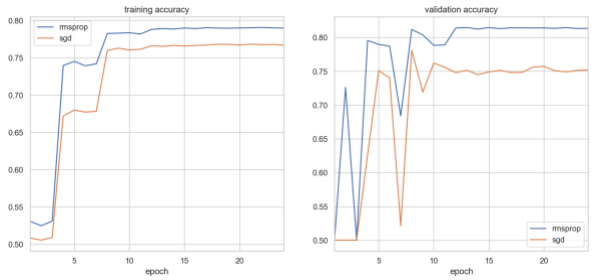

Figure 11: Accuracy of training and validation

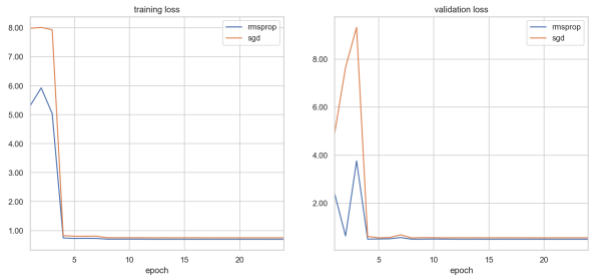

Figure 12: Loss of training and validation

An interesting fact is that it seems that only 12 or 13 epochs were needed to get a validation accuracy above 80%.

3.2.5 Tests using the testset

After having these numbers both networks were tested using the testset. Instead of accuracy the ROC and AUC were used as this is the final metric of interest.

For this test the file test_inception.py was used. The reason for using a separate file was simple: As the duration of training and validation was unknown in advance the model was stored at the end of training and loaded in this separate file.

Testing was done in these steps:

- Define the data transforms.
- Load the test dataset.
- Create the dataloader for this dataset.
- Create the model and load the saved model.
- Test the model.
- Write the results into a csv file.

3.2.5.1 Data Transforms

No image transformations were done for the test set as they need be pass the network without changes. As usual they were only

resized, changed to a tensor and normalized by mean pixel subtraction per channel.

3.2.5.2 Data loading

Data loading was done the same way as for training and validation.

3.2.5.3 Model creation

As for training and validation the code for the inception network was used that is part of PyTorch. After that the saved model was loaded using the functions load_state_dict of the model and torch.load. The latter loads the file from disc, the former loads the weights into the model.

3.2.5.4 Test the models

The test run is only executed once instead of multiple epochs. The code in PyTorch is pretty much the same as using validation:

```
(1)  model.eval()
(2)  with torch.no_grad():
(3)    for images, labels in testloader:
(4)      y_preds = model(images)
(5)      values, _ = torch.max(y_preds.data, 1)
```

In line 5 only the values (probabilities per class) are stored. These are saved to a Python list. This is because instead of calculating accuracy using the labels the probabilities must be used to calculate ROC and AUC.

3.2.6 Results (ROC and AUC)

The csv files produced by test_inception.py were used to calculate ROC and AUC. For these calculations the Python module sklearn.metrics from the scikit-learn project[7] contains the functions roc_curve and roc_auc_score. Both need the input to be the truth labels and the predictions made by the network.

The results again show that the first network is trained more accurate than the second one.

As mentioned in the overview section the scores for AUC are

- 0.87 (first model)
- 0.85 (second model)

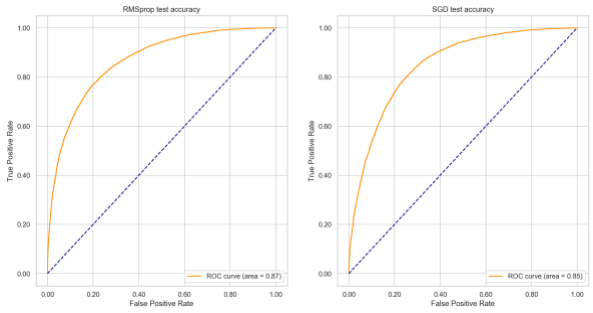

Figure 13: ROC and AUC of both models

3.2.7 Creation of CNN and G-CNN

The input images were normalized with mean and standard deviation of the Imagenet dataset before they were fed through the pretrained inception networks. This technique will be used as well when training the own CNN and G-CNN.

These statistics were computed using the function utils.data.get_mean_std in the file utils/data.py.

3.2.7.1 CNN

For an additional reference a CNN was created that was trained from scratch. The network design was mainly influenced by [24]. This paper has shown that pooling layers can be replaced by convolutional layers with increased stride and fully connected layers can also be replaced by simple 1-by-1 convolutions.

The CNN was therefore modelled using the *Large All-CNN Model for CIFAR-10* proposed in the paper (Appendix A). The model from bottom to top including **N** as batchsize was:

```
(1)  Input: Tensor(Nx3x96x96)
(2)  2x2 conv, stride 1x1 > LeakyRelu: Nx320x95x95
(3)  2x2 conv, stride 1x1 > LeakyRelu: Nx320x94x94
(4)  2x2 conv, stride 2x2 > LeakyRelu: Nx320x47x47
(5)  Dropout p=0.1
(6)  2x2 conv, stride 1x1 > LeakyRelu: Nx640x46x46
(7)  Dropout p=0.1
(8)  2x2 conv, stride 1x1 > LeakyRelu: Nx640x45x45
(9)  2x2 conv, stride 2x2 > LeakyRelu: Nx640x22x22
```

[7] https://scikit-learn.org

(10) Dropout p=0.2
(11) 2x2 conv, stride 1x1 > LeakyRelu: Nx960x21x21
(12) Dropout p=0.2
(13) 2x2 conv, stride 1x1 > LeakyRelu: Nx960x20x20
(14) 2x2 conv, stride 2x2 > LeakyRelu: Nx960x10x10
(15) Dropout p=0.3
(16) 2x2 conv, stride 1x1 > LeakyRelu: Nx1280x9x9
(17) Dropout p=0.3
(18) 2x2 conv, stride 1x1 > LeakyRelu: Nx1280x8x8
(19) 2x2 conv, stride 2x2 > LeakyRelu: Nx1280x4x4
(20) Dropout p=0.4
(21) 2x2 conv, stride 1x1 > LeakyRelu: Nx1600x3x3
(22) Dropout p=0.4
(23) 2x2 conv, stride 1x1 > LeakyRelu: Nx1600x2x2
(24) 2x2 conv, stride 2x2 > LeakyRelu: Nx1600x1x1
(25) Dropout p=0.5
(26) 1x1 conv, stride 1x1 > LeakyRelu: Nx1920x1x1
(27) Dropout p=0.5
(28) 1x1 conv, stride 1x1 > LeakyRelu: Nx2x1x1
(29) Reshape: Nx2x1
(30) ReduceMean: Nx2
(31) Softmax

As seen the design is exactly the same of the model in the paper. The two steps called 'Reshape' and 'ReduceMean' is what global average pooling would do in a traditional CNN.

The network is initialized using Glorot initialization as this is what is done in the Caffe model used in the original paper.

3.2.7.2 G-CNN

The G-CNN was modelled exactly like the CNN but the first layer (conv1) is replaced by a P4ConvZ2 layer and all convolutional layers are replaced by P4ConvP4 layers.

These layers do rotate their filters. Mirrors and reflections are not included.

The implementation of these layers was done in [15] and was installed before training.

3.2.7.3 Loss function

For both models CrossEntropyLoss was chosen again.

3.2.7.4 Optimizer

For comparison RMSprop was chosen again. The parameters used for both models were:

Learning Rate	0.001 – 0.1
Momentum	0.9
Epsilon	0.0000001
Weight Decay	0.9

Epsilon was set to a small value as this is done by default. As mentioned above it was set to 1.0 for Inception as this was needed.

Weight Decay and Momentum are left the same as in the transfer learning cases.

3.2.7.5 Learning Rate Scheduler

As seen above a different learning rate scheduler was chosen. According to [21] a cyclic learning rate can help converging faster. As only 24 epochs should be trained again the scheduler chosen was therefore CyclicLR. This scheduler switches the learning rate per default every 2000 iterations between min and max learning rate given above.

3.2.7.6 Image preprocessing

As the image augmentations used during transfer learning seem not be helpful it was chosen to switch to another Python package named imgaug. This is more flexible and provides more methods for augmentation.

Validation and testing sets were not augmented again. For training the augmentations chosen were

- With probability of 0.3 rotate between -90 and 90 degrees.
- With probability of 0.25 horizontally mirror.
- With probability of 0.25 vertically mirror.
- With probability of 0.5 either multiply all pixels randomly in range of 0.5 to 1.5 and change the contrast in the same range **or** change hue and saturation and add gaussian blur.
- For every image randomly dropout pixels (zero out) with probability between 0 and 0.2.
- transform images by moving pixels locally around using displacement fields.

3.2.7.7 Training

Both networks were trained at the same time on a Google Cloud VM having 2 Tesla T4

GPUs attached. One model was trained on the first GPU, the other one on the second.

Full code is stored in the files:

- gcnn/helpers.py: To separate all code of the various steps described above the same helper functions of the CNN part were split in this file into train_one_epoch and validate_model.
- gcnn/models.py: This holds the PyTorch models AllConvNet and AllGConvNet.
- utils/data.py: A new class ImgAugTransform is stored here that is used to transform the input data using Python package imgaug. In addition the function get_mean_std that calculates mean and standard deviation of a dataset is placed here.
- train_allnet.py: This is the main file that trains, validates and tests both all convolutional networks.

3.3 Refinement

3.3.1 Memory consumption

The first training was not successful at all because both models consumed way too much memory on GPU (each GPU has 16GB RAM). The fixes for that were:

- Transfer learning was possible using a batch size of 200. This was reduced to batch size of 64. (This is the batch size used as well in [13].)
- The All-Gconv-CNN was still to memory hungry. Hence the number of layers was reduced from 17 to 9. (This is the initial size as well in [24].)

3.3.2 Training time & Accuracy

In [14] it is stated that a neural network is trainable in a day using the PCam dataset. This can not be confirmed using the implementation of this paper. Training time of one epoch took around **2.5 hours** for the G-CNN, the CNN runs for **2 hours** . Due to that the initial plan to evaluate G-CNN with a fully trained CNN was given up (this was not part of the proposal anyway).

In addition the first tests have shown that the network was not learning at all.

Several steps were taken to improve training time and accuracy:

- To speed up memory copy from host to GPU the dataloader was configured to pin the data to locked memory.
- Using pinned memory PyTorch can use copy and training asynchron by adding the option non_blocking=True when copying the data from host to GPU.
- The number of layers of the G-CNN was reduced again to only 6 layers.
- Weight initialization was changed from xavier_uniform_ to kaiming_normal_.
- The activation function was changed from leaky_relu to relu.
- Image augmentation was removed.
- The learning rate scheduler was changed to StepLR again.
- Different learning rates were tried.
- Instead of RMSprop the Adam optimizer was tried.
- As a code bug was suspected that prevents optimizer.step() from optimizing the weights, training and validation were merged into one function again. This was used during transfer learning as well.
- Network layout was switched from convolutional layers only to include two fully connected layers as well as max pooling layers (these are named plane_group_spatial_max_pooling to support the group equivariance).

The intermediate code was updated in main.py, models.py, helpers.py. The G-CNN used only 6 layers, channels increasing from 96 to 384.

Unfortunately no change mentioned above helped reducing training time. Perhaps the Google Cloud is the issue, maybe the GPU,

possibly there is a bug in the code or network design.

Therefore a full training cycle was run for about 3 days because up to that point only a few epochs were run on every test.

This test show a random model. Accuracy was around 50% all the time for 24 epochs (what means random guessing in a binary classification problem I guess).

Finally the "normal" group equivariant neural network that includes full connected layers and max pooling was tested on the CIFAR10 dataset. This was done to see whether the network is able to learn at all. The layout of the network is

(1) Input: Tensor(Nx3x96x96)
(2) 3x3 p4convz2, stride 1x1 > Relu: Nx384x94x94
(3) 3x3 p4convp4, stride 1x1 > Relu: Nx384x92x92
(4) Max_Pooling: Nx384x46x46
(5) Dropout p=0.2
(6) 3x3 p4convp4, stride 1x1 > Relu: Nx768x44x44
(7) 3x3 p4convp4, stride 1x1 > Relu: Nx768x42x42
(8) Max_Pooling: Nx768x21x21
(9) Dropout p=0.3
(10) 3x3 p4convp4, stride 1x1 > Relu: Nx1536x19x19
(11) 3x3 p4convp4, stride 1x1 > Relu: Nx1536x17x17
(12) Reshape: Nx443904
(13) Linear > Relu: Nx1024
(14) Dropout p=0.4
(15) Linear: Nx2

As seen in the computational graph in the companion jupyter notebook P4ConvP4 rotates the kernel as expected and produces a four times larger output (as the filter has four "positions"). The code for the first layer specified a kernel size of 96, but the output of this layer is 384 large.

The graph shows that computation using this implementation is very complex and involves a lot of internal functions on Tensors. This is one component that is suspected to contribute to the large runtime.

During the project the Keras implementation was not tested to verify that.

This model was able to learn, but very slow. It achieved an accuracy of **64%** on the testset. This might be tuned using changed parameters. But this was not done in the project because this test was only made to see whether the network can learn.

3.3.3 Debugging

As it was known now that the model is working further investigation was done.

3.3.3.1 Model output on single Tensor

When creating the model and just sending a single tensor through the network one can see a single prediction. For example:

```
(1) import gcnn
(2) import torch
(3) allcnn = gcnn.AllConvNet(3,2)
(4) print(allcnn(torch.zeros([1, 3, 96, 96])))
```

This simple step returned a Tensor with to values that are equal to 0.5 every time!

```
(1) tensor([[0.5000, 0.5000]], grad_fn=<SoftmaxBackward>)
```

Printing the value of x in forward() has shown that all values were set to zero. This was due to one error: In __init__ the bias values of the networks were set to 0. This was done as the Caffe model of the All-Convolutional-Model was using that too. After removing this line the model returned random values as expected. In addition screening the source code of the layer used one can see that the weights are initialized by kaiming_uniform_. Due to that the initialization block is completely removed.

The second important error seen here was that the implementation of forward() regarding the dimensions might be wrong. The dimension of the output of the last Conv-Layer after applying LeakyRelu for the "normal" Convolutional network is

```
(1) torch.Size([1, 2, 1, 1])
```

In contrast the G-CNN produces

```
(1) torch.Size([1, 2, 4, 21, 21])
```

In the CNN the last layer of adaptive_avg_pool2d was replaced with reshape() and mean(). That is was this layer is doing. It is unknown how to replace this layer in a G-CNN.

Due to that the idea of using an All-Convolutional-Network was dropped.

3.3.3.2 HDF5 Dataset/Dataloader

To investigate whether the non-threadsafe HDF5 library was the bottleneck all images contained in the dataset were stored as PNG files on disk. The Python file save_images.py was used for that.

3.3.3.3 Code-Bugs

In addition to the fixes above following code bugs were found:

- The learning rate scheduler was not called after every batch but only after each training epoch.

3.3.4 Final code changes

The code was adapted with all changes mentioned above to test the Group-Equivariant network again. The model used for CIFAR10 was used for that with a small change to save memory: The number of channels referenced above was halved. All changes summarized:

- All images were extracted from the HDF5 files to disk. The folders are organized into
  - root/train/normal
  - root/train/tumor
  - root/valid/normal
  - root/valid/tumor
  - root/test/normal
  - root/test/tumor
- That way using ImageFolder dataset it is ensured that normal gets label **0** and tumor gets label **1**. This can be seen by calling datasets['train'].class_to_idx which shows {'normal': 0, 'tumor': 1}.
- These images were moved to a faster disk (from GCP standard to an SSD).
- The dataloaders use pinned memory for faster transfer of the images to GPU RAM.
- As validation and testing phases do not use the autograd feature the initial batchsize used for training is doubled for them.
- The GPU uses non-blocking operations so that dataloader and network can work in parallel.
- A G-CNN model was used that includes linear and max pooling layers instead of an All-Convolutional-Network.
- This network does not initialize its bias parameters to zero anymore.
- The weights are initialized with the default method.
- The number of channels of each layer was halved to prevent CUDA out of memory errors.
- Like in the test with CIFAR10 Adam was used as optimizer and was configured with default parameters except the learning rate. Weight Decay was not used as this is implemented as L2 regularization. Compare with [25].
- This LR scheduler steps now after each batch and not after each epoch.
- CrossEntropyLoss as loss function is configured to calculate the sum of the loss over each batch instead of the mean. This is to calculate better statistics.
- The learning rate scheduler is the same as during the transfer learning.
- As the G-CNN rotates the kernel the rotations during image augmentation was dropped.

In the initial test the length of one training epoch dropped from **2.5 hours to 30 minutes**!

The final parameters used:

Learning Rate	0.05
LR Decay	LR * 0.1 per 20 Epochs
Epochs	100
Batch-Size	64 (Training) 128 (Valid and Test)

This learning rate was chosen to start with a higher learning rate that is going to be decayed 4 times during training.

The shown batch size was chosen due to memory constraints on the GPU. Larger batches ran out of memory. As autograd is disabled during validation and test a higher batch size is possible here.

Unfortunately the model was still not able to learn during the last test and stuck around 50% accuracy.

4 Results

4.1 Model Evaluation and Validation

As the task was not solved there is no final result to be shown. The bad performance of the G-CNN over 24 epochs was:

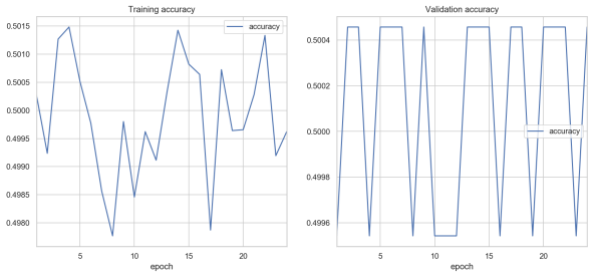

Figure 14: G-CNN random accuracy

The loss was

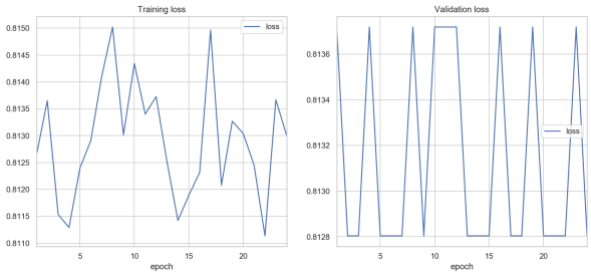

Figure 15: G-CNN random loss

This was not higher during the final test, hence not statistics of this run is shown here.

In contrast it was seen that the network itself using the parameters shown above was able to learn on the CIFAR10 dataset.

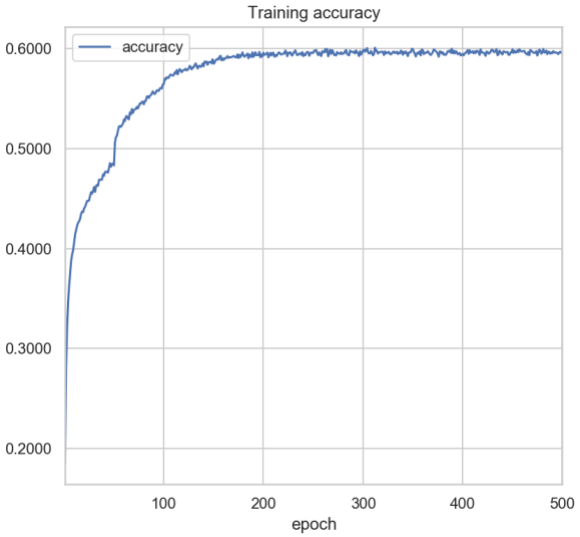

Figure 16: G-CNN training accuracy on CIFAR10

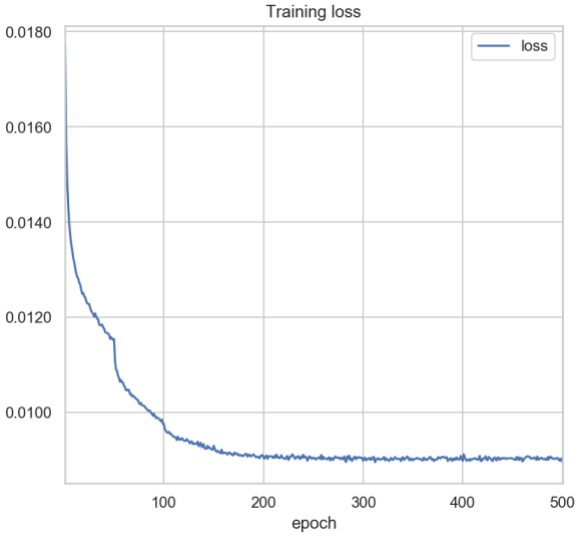

Figure 17: G-CNN training loss on CIFAR10

The assumption is that the model chosen is just too simple to learn the features of the input PCam dataset. It was able to learn some features of natural images of the CIFAR10 dataset but here the accuracy stuck around 60%.

In addition deeper models were not tried because 16 GB RAM on GPU was not enough for some test cases. The guess is that

the implementation of GrouPy [8]is too complex to hold all gradients during backward propagation updating the autograd graph.

4.2 Justification

The final model does not solve the task and has shown a bad performance.

The base benchmark of PCam was implemented as G-CNN as well. It was modelled as DenseNet consisting of 5 Dense Blocks alternated with Transition Blocks. This is shown in [13].

The second benchmark was implemented as Inception v3 network trained using transfer learning. Without further tweaking this has shown a lower performance than the base benchmark but has a good result of AUC 0.87. The base benchmark has shown an AUC of 0.96.

5 Conclusion

5.1 Free-Form Visualization

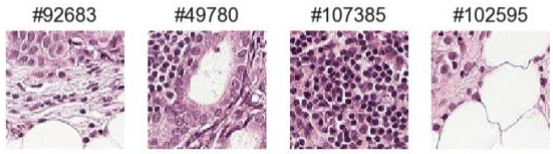

Figure 18: Input images – tumor

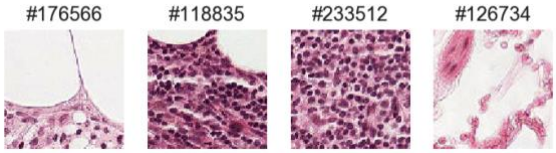

Figure 19: Input images – normal

Provided above are samples of input images of the training set. The first images contain tumor pixels the second row contains images without tumor pixels.

As stated above the guess about the bad performance is that the network trained is just too simple to detect relevant features of these input images. Even an inception network (that is a very deep network) using transfer learning is not able to achieve a high accuracy and AUC without further tweaking.

A hard problem in image classification even on natural images is the similarity between objects shown. Inception is trained on a large number of images showing natural objects. The images in the chosen dataset are all very similar as they show patches of biological tissue scanned from microscope slides.

The theory is that it needs a different architecture that is deeper and perhaps using different layers (and combination of layers) to detect special features of these images.

5.2 Reflection

During this project:

- A public dataset containing patches of Whole-Slide-Images produced by Digital Pathology was found.
- This dataset was made accessible using custom Python classes that can be used by PyTorch.
- These images were used to train an initial Neural Network using transfer learning.
- Based on the results of this network a second network was trained with different settings.
- Having three benchmarks now available three networks were created from scratch:
    - One All-Convolutional-Network
    - One All-Group-Convolutional-Network
    - One Group-Convolutional-Network
- The whole training process was then tuned to get higher training performance and accuracy.

The whole process was difficult:

[8] https://github.com/adambielski/GrouPy

- This was the first time I worked with HDF5 files. Understanding the content, how it is organized and how it can be accessed took a bit time. This is shown in the Jupyter-Notebook.
- A lot of time was spent learning how PyTorch works. This was not only difficult but interesting too. PyTorch is a very powerful framework I'll further work with in the future.
- The most challenging part was how to design a Group-Equivariant Convolutional Network as I had to learn how G-CNNs work. This part has shown that my understanding of Convolutional Neural Networks was not complete. I had to learn a lot more how padding, strides and the kernel and the math works. The resource that helped me the most is [26].
- The two most interesting parts were
    - how to setup and run the code on Google Cloud and
    - how to start debugging a neural network and the process of training.

I was very pleased to see that I was able to decrease the training time per epoch.

But in contrast I am very dissatisfied that the model I created was not able to learn at all. I expected that accuracy would increase.

5.3 Improvement

There are a lot of improvements to think off:

- The model could be trained on multiple GPUs to decrease training time.
- The math of Group theory could be implemented in C++ in PyTorch so that GConv-Layers and Pooling layers are directly implemented and available in PyTorch. The computational graph in the Jupyter Notebook shows that GroupPy needs to track a lot of variables and hence gradients. That makes the model slower.
- Instead of designing and training a whole (G-)CNN from scratch experiments can be made using transfer learning. Different models (like DenseNet) can be used and fine tuned instead of used as fixed feature extractors.
- Transfer learning can be trained for more than the 24 epochs by using the extracted images instead of the HDF5Matrix created.
- The code can be refactored to be more modular and can be further enhanced by using checkpointing. That can ensure that an interrupted training process can be restarted.
- The model chosen should be further debugged to see why it was not able to learn, for example the features learned can be visualized (what I currently do not know how).
- One paper I would like to implement is described in [27].

6 Technical environment

All tests were made using Google Cloud Platform using a VM having 4 vCPUs, 15 GB of memory (n1-standard-4) and one or two NVIDIA Tesla T4 GPUs attached. The image used was a Deep Learning Image having PyTorch 1.1.0 and fastai m27 pre-installed with conda that is optimized with CUDA 10.0 and Intel MKL-DNN.

7 References

[1] Y. Liu, K. Gadepalli, M. Norouzi, G. E. Dahl, T. Kohlberger, A. Boyko, S. Venugopalan, A. Timofeev, P. Q. Nelson, G. S. Corrado, J. D. Hipp, L. Peng and M. C. Stumpe, "Detecting Cancer Metastases on Gigapixel Pathology Images," 2017. [Online]. Available: https://arxiv.org/abs/1703.02442.

[2] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A.

v. d. Laak, B. v. Ginneken and C. I. Sánchez, "A Survey on Deep Learning in Medical Image Analysis," 2017. [Online]. Available: https://arxiv.org/abs/1702.05747.

[3] B. E. Bejnordi, G. Litjens, N. Timofeeva, I. Otte-Höller, A. Homeyer, N. Karssemeijer and J. A. v. d. Laak, "Stain Specific Standardization of Whole-Slide Histopathological Images," *IEEE Trans. Med. Imaging*, no. 35 (2), pp. 404-415, 2016.

[4] D. Wang, A. Khosla, R. Gargeya, H. Irshad and A. H. Beck, "Deep Learning for Identifying Metastatic Breast Cancer," 2016. [Online]. Available: https://arxiv.org/abs/1606.05718.

[5] R. Chen, Y. Jing and H. Jackson, "Identifying Metastases in Sentinel Lymph Nodes with Deep Convolutional Neural Networks," 2016. [Online]. Available: https://arxiv.org/abs/1608.01658.

[6] T. S. Cohen and M. Welling, "Group Equivariant Convolutional Networks," 2016. [Online]. Available: https://arxiv.org/abs/1602.07576.

[7] "Genomic data commons data portal (legacy archive).," [Online]. Available: https://portal.gdc.cancer.gov/legacy-archive/search/f. [Accessed 23 06 2019].

[8] "Biospecimen Research Database," 23 06 2019. [Online]. Available: https://brd.nci.nih.gov/brd/image-search/searchhome.

[9] "Stanford Tissue Microarray Database," 23 06 2019. [Online]. Available: https://tma.im/cgi-bin/home.pl.

[10] "Ovarian Carcinomas Histopathology Dataset," 23 06 2019. [Online]. Available: http://ensc-mica-www02.ensc.sfu.ca/download/.

[11] "The CAMELYON16 challenge," [Online]. Available: https://camelyon16.grand-challenge.org/. [Accessed 23 06 2019].

[12] "The CAMELYON17 challenge," [Online]. Available: https://camelyon17.grand-challenge.org/. [Accessed 23 06 2019].

[13] B. S. Veeling, J. Linmans, J. Winkens, T. Cohen and M. Welling, "Rotation Equivariant CNNs for Digital Pathology," 2018. [Online]. Available: https://arxiv.org/abs/1806.03962.

[14] "The PatchCamelyon (PCam) deep learning classification benchmark.," [Online]. Available: https://github.com/basveeling/pcam. [Accessed 23 06 2019].

[15] A. Bielski, "GitHub," [Online]. Available: https://github.com/adambielski/GrouPy. [Accessed 25 06 2019].

[16] S. V. a. L. Lab, "CS231n: Convolutional Neural Networks for Visual Recognition," [Online]. Available: http://cs231n.github.io/convolutional-networks/. [Accessed 25 06 2019].

[17] T. Cohen, "GitHub," [Online]. Available: https://github.com/tscohen/GrouPy. [Accessed 25 06 2019].

[18] B. Veeling, "GitHub," [Online]. Available: https://github.com/basveeling/keras-gcnn. [Accessed 25 06 2019].

[19] S. Chilamkurthy, "PyTorch Tutorials," [Online]. Available: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html. [Accessed 26 06 2019].

[20] N. Inkawhich, "PyTorch Tutorials," [Online]. Available: https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html. [Accessed 26 06 2019].

[21] L. N. Smith, "Cyclical Learning Rates for Training Neural Networks," 2015. [Online]. Available: https://arxiv.org/abs/1506.01186.

[22] L. N. Smith and N. Topin, "Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates," 2017. [Online]. Available: https://arxiv.org/abs/1708.07120.

[23] E. Waite, "PyTorch Autograd Explained - In-depth Tutorial," [Online]. Available: https://www.youtube.com/watch?v=MswxJw-8PvE. [Accessed 27 06 2019].

[24] J. T. Springenberg, A. Dosovitskiy, T. Brox and M. Riedmiller, "Striving for Simplicity: The All Convolutional Net," 2014. [Online]. Available: https://arxiv.org/abs/1412.6806.

[25] L. Xu, "GitHub," [Online]. Available: https://github.com/egg-west/AdamW-pytorch. [Accessed 30 06 2019].

[26] O. U. Florez, "GitHub," [Online]. Available: https://github.com/omar-florez/scratch_mlp/. [Accessed 01 07 2019].

[27] P. Follmann and T. Bottger, "IEEE Xplore Digital Library," 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8354195.