Re-Implementation of Shallow Convolutional Networks for Predicting Saliency Maps

1st Finn Wilkinson
dept. Computer Science
University of Bristol
Bristol, United Kingdom
fw17231@bristol.ac.uk

2nd Thomas Raybould
dept. Computer Science
University of Bristol
Bristol, United Kingdom
tr17544@bristol.ac.uk

I. Introduction

Representing the probability of visual attention, saliency is an important part of many computer vision applications, such as automatic image cropping [5] and image captioning [6]. Traditionally, computing saliency maps has been achieved using hand crafted features, however, *Pan et al* [1] propose a solution that uses a completely data driven method instead; a shallow convolutional neural network (CNN) using end-to-end learning.

Large amounts of data are required to train the network, but with annotated datasets such as SALICON, iSUN, and MIT300, we have the necessary resources available to us. As a saliency map must display the saliency value for each pixel of an image, rather than a single score, we also require that predicted values transition smoothly between neighbouring pixels in order to form an accurate distribution.

In this paper, we will implement a shallow CNN trained using the SALICON dataset in order to produce saliency map predictions. We discuss and implement the architecture presented by *Pan et al*, as well as show our successfully replicated results. Additionally, we aim to suggest an improvement to their method that increases the accuracy of evaluation metrics and the predicted saliency maps.

II. Related work

This section reviews some more recent papers that also discuss saliency prediction using convolutional neural networks.

One more recent attempt at improving the prediction attained by a CNN was proposed by *Cornia et al* [3] in 2018, where the authors incorporate neural attention mechanisms to iteratively refine their predictions. Their solution, SAM (Saliency Attentive Model), is composed of three main architecture components.

First, there is the Dilated Convolutional Network, or a Dilated CNN, that extracts the feature maps from the input image. As with our implementation, the size of the feature maps are reduced, however, the use of a Dilated CNN limits the negative effects attributed to drastic re-scaling, as the maps are only need to be re-scaled by a factor of 8. This increase in resolution helps the model to predict more accurate results, without incurring a performance hit due to the additional data it must process.

Next is the Attentive Convolutional Long Short-Term Memory (LSTM) module. Here, different locations of the the saliency feature map are repeatedly processed in order to enhance them. Initially, an attention map is generated by combining the previous hidden state and the input. Once this is normalized through a softmax operator, the attention map is applied to the input with an element-wise product. This resulting stack of features is then iteratively fed to the LSTM, and after a fixed number of iterations, the last hidden state is taken as the output of this module.

The final section of the SAM architecture is a Learned Priors Module, which combines the output of the LSTM with learned priors in order to model the centre bias present in human-eye fixations [3]. Each of the priors used is a 2D-Gaussian function whose mean and co-variance matrices are freely learn-able by the network itself, meaning the priors are inferred from the ground-truth data and doesn't have to rely on assumptions from separate biological studies.

Results from this alternate implementation were very promising on the 2017 SALICON dataset, which replaces the velocity based fixation detection algorithm used to attain more eye-like fixations [3]. The SAM model yielded better results than DeepGazeII, SalNet, and ML-Net implementations on all metrics bar AUC, where it fell short of DeepGazeII by 0.002, therefore showing that incorporating dilated convolutions, an attentive mechanism and learned prior maps into a standard CNN architecture provides improved results over existing models in the saliency prediction field.

Traditionally, models for saliency prediction are trained using a loss function. In 2018, a paper by *Pan et al* used adversarial training instead [4]. Adversarial training involves two networks; a generator network and a discriminator network. In this paper, SalGAN was the generator network, and it produced saliency map predictions. The goal of the discriminator network is to determine whether a saliency map is real or if it was generated by SalGAN. The goal of SalGAN is to produce saliency maps that are indistinguishable from real saliency maps.

The filter weights in SalGAN were trained using a combination of content loss and adversarial loss, where content loss does pixel-wise comparison between the ground truth and prediction, and adversarial loss depends on the real or synthetic

prediction of the discriminator network.

Pan et al, 2018 found that a BCE-based content loss was effective when used to initialise the saliency prediction network, and as regularization for stabilizing the adversarial training. They also found that adversarial training improved all bar one saliency metric when compared to training without.

III. Dataset

For training and validation of our model, we will be using the SALICON dataset, which provides saliency annotations on 10,000 images from the MS COCO database. The saliency annotations were produced using a new paradigm where a computer mouse could be used to track viewing behaviour [2]. Horizontal flip data augmentation was performed as a pre-processing step and thus is not included in our implementation. Overall, we have 20,000 training images and for validation we have 500 separate images. The training data is stored in a pickle file train.pkl, and the validation data in val.pkl. Pickle is a python module that serializes and de-serializes an object's structure in order to store it on disk [13]; we use pickle to store both the dataset and our predictions produced by the network. Our training labels are the ground truth saliency maps for the images in the SALICON dataset.

IV. Input

Our model takes images and their associated ground truth saliency maps as input. A saliency map is a probability distribution that represents the likely-hood of visual attention for each part of an image. Traditionally, data would be collected manually or via crowd-sourcing by assessing the visual attention of people looking at images (via eye tracking or mouse click/movement) [2] and averaging this data produces a saliency map. Results from a study carried out by *Underwood et al* shows that people pay more attention to, and are quicker to look at, areas in an image with high colour variation/contrast and greater visual complexity [7]. Despite their work being on people studying for a memory test on objects in images, these conclusions can also be seen in our ground truth saliency maps from Figure 1. We can clearly see that areas with more attention (coloured in yellow) correspond to the complex, vibrant, and contrasting colour sections of the image. We can also see that focus is drawn to other key areas, such as faces, text, and any numbers present in the images. After the training of our CNN, we hope to see these kind of areas in the input images recognised as places with high visual attention. This would result in the final saliency output maps having similar high intensity areas as the ground truth labels it is given.

V. Shallow Architecture (Pan et al)

The shallow architecture described by *Pan et al* consists of three convolutional layers and two fully connected layers. Table I shows details of each layer in the network. Input is a [96 x 96] RGB image, hence the size given in the table. The dimension of the convolutional layers corresponds to the *kernel size* x *output channels*. For example, the first convolutional layer uses a [5 x 5] kernel and has 32 output

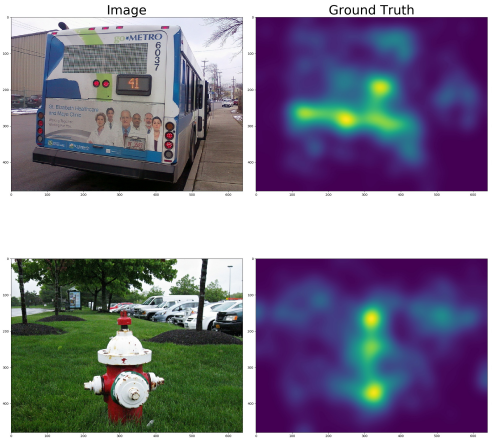

Fig. 1: Sample Input Images and Their Ground Truths

channels. Each convolutional layer is followed by a rectified linear unit non-linearity (ReLU) layer, and then a max pooling layer. Images are reduced from a [96 x 96] input to [10 x 10] output after the final max pooling layer. The first fully connected layer produces a 4608-dimensional vector, which is sliced into two 2304-dimensional vectors and then passed through a max out layer, that computes the max value between adjacent pixels. The final output of the network is a 2304-dimensional vector, which when re-formatted to a [48 x 48] image, gives us the predicted saliency map for the input image.

Shallow Architecture (*Pan et al*)
Input (96 x 96 x 3)
Convolution 1 (5 x 5 x 32)
Max Pool 1 (kernel = (2 x 2), stride = 2)
Convolution 2 (3 x 3 x 64)
Max Pool 2 (kernel = (3 x 3), stride = 2)
Convolution 3 (3 x 3 x 32)
Max Pool 3 (kernel = (3 x 3), stride = 2)
Fully Connected Layer 1 (4608)
Slice &
Max Out Layer
Fully Connected Layer 2 (2304)

TABLE I: Details of Shallow Architecture

VI. Implementation Details

In this section, we will detail how we implemented and trained the shallow network in order to successfully replicate the results by *Pan et al*. The implementation differs from that by *Pan et al* [12] since we are using the PyTorch library, rather than the deep learning libraries Theano and Lasagne.

A. Loss Function

Pan et al do not specify the loss function they used for training their network in their paper, however, we are able to see from the Lasagne documentation [11] that the default loss function computes the element-wise squared error between the predictions and ground truths. We decided to use *mean squared error* as this is easily implemented using PyTorch,

and is essentially computing the same loss; we are minimizing the average per-pixel error, whilst *Pan et al* minimize the total per-pixel error.

B. Optimiser

We optimised training using *Stochastic Gradient Descent* (SGD) with momentum. Initially the learning rate is set to 0.03, however, in order to replicate the results by *Pan et al* we needed to reduce the learning rate as the training progresses. This was done by pre-computing the learning rates for each epoch by creating an array of length 1000 (the number of epochs) with evenly spaced values over the interval $[0.03, 0.0001]$. We then updated the learning rate parameter at the start of every epoch in the training. Like *Pan et al*, we used the Nesterov momentum method, which improves training by computing gradient with respect to approximate future parameters [8]. Weight decay was initialised to 0.0005, and is not changed at all during training.

C. CNN

To implement the shallow architecture from the *Pan et al* paper, we defined a CNN class with three functions. The initialisation of the class defines the convolutional, fully connected, and max out layers of the network. We also initialise the weights and biases of these layers. Layer weights are initialised by a normal distribution with $\mu = 0.0$, $\sigma = 0.01$, and layer bias is set to 0.1. Table II shows the parameters for each of the three convolution layers, while parameters for the max pooling layers can be seen in Table I. It is necessary to include zero padding in the convolutional layers so that the dimension of the output is correct, such that the max pooling layers can work correctly with their given stride and kernel size. A consequence of this is that our final output from the max pooling layer gives an [11 x 11] image rather than [10 x 10], however, this does not effect the accuracy of our model as this increase in output size is just our padding.

Conv. Layer	**Kernel**	**Padding**	**In**	**Out**
1	5 x 5	2	3	32
2	3 x 3	1	32	64
3	3 x 3	1	64	128

TABLE II: Parameters of Convolution Layers

Implementing the forward pass and defining how data is passed through the network is in the forward function of the CNN class. Output from each convolution layer gets passed through a ReLU layer before a max pooling layer. After the convolutional layers, we need to flatten the images into one dimension, so that we can pass them through the fully connected layers. Data is now a 15488-dimensional vector, which comes from the [11 x 11] image and 128 channels being flattened into one dimension. The first fully connected layer takes the 15488-dimensional vector and reduces it to a 4608-dimensional vector.

Before the second fully connected layer, the network performs a split and max out operation on the data. We split the data such that adjacent pixels are separated, so that the max out layer performs the max between them. This reduces the size of our vector to [2304], which we then pass through the final fully connected layer to get our output vector; also of size [2304], representing the [48 x 48] saliency map.

D. Trainer

Training, optimising, validation, and computing the predictions is controlled by the trainer class. The first thing we do is tell the model that we are in training mode so that it learns the layers. For periodic validation, we put the model in evaluation mode, which means the model will not learn for any of the layers. We train the network for 1000 epochs, using a batch size of 128.

As mentioned previously, we update the learning rate for every epoch. First, we compute a forward pass, which produces the predictions. Next, we calculate the loss between the current predictions and the ground truths, which is used to compute the gradient. Using this gradient we update the parameters of the model. PyTorch accumulates gradients for subsequent backwards passes, so we need to reset gradients to zero after updating our parameters. If we didn't do this, the gradient would be incorrect throughout the training.

For periodic validation, we used a separate image set of 500 images. We put the model in evaluation mode, so that the model won't learn, and calculate the average loss for all 500 images, then logging this. Training loss is logged every five steps, and validation is done every five epochs. There are 157 steps per epoch, which give us 31 training points per epoch.

Final predictions are stored as a 2304-dimensional vector in the file preds.pkl. Running the visualisation script resizes the vector into the [48 x 48] saliency map, and produces images like that seen in Figure 5. To obtain the accuracy of our model, we ran the evaluation script on the predictions and ground truths.

The model parameters get saved to the file checkpoints.pt every 50 epochs, which allowed us to get predictions from the model without needing to retrain. This also meant that we could resume training from the epoch corresponding to the saved model by reloading the checkpoint data. This can be done by specifying the path to the checkpoint data and which epoch to start training from.

VII. Replicating Quantitative Results

Our implementation successfully replicated the results produced by *Pan et al*, taking approximately 3 hours to train. Table III shows our results across three metrics: CC, AUC Shuffled, and AUC Borji. For each of these metrics, our implementation saw a marginal increase of 0.01 when compared to the target results.

SALICON (val)	CC	AUC Shuffled	AUC Borji
Target Results	0.65	0.55	0.71
Our Results	0.66	0.56	0.72

TABLE III: Our validation results and the result targets using the SALICON dataset.

VIII. Training Curves

Over-fitting is problematic as it means the model isn't generalised enough to be able to make good predictions for unseen data. In order for us to be able to gauge whether or not our model is over-fitting, we tracked the training and validation loss during the model training. The blue line in Figure 2 shows the training loss over the 1000 epochs; the orange line is the validation loss. The X axis is the step number (157 steps per epoch), and the Y axis is the loss value.

From Figure 2, you can see that the model is starting to stop learning at approximately 40,000 steps; during epoch 254. The training loss generally tends slowly downwards, as does the validation loss, however, the separation between the two curves gradually starts to grow. This tells us that the model is over-fitting, as we see more loss between our predictions and ground truths when exposing the model to new images. This isn't so clear from Figure 2 due to the training loss' erratic behaviour, but smoothing the curves out, seen in Figure 3, shows a clear, gradual separation between the lines.

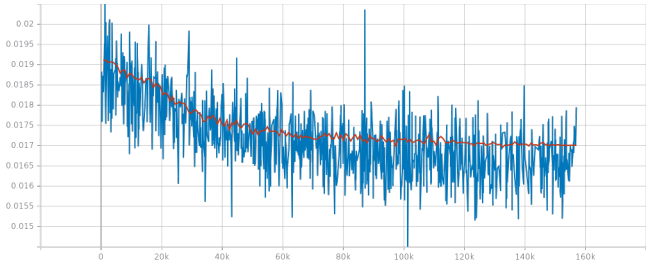

Fig. 2: Training loss curve

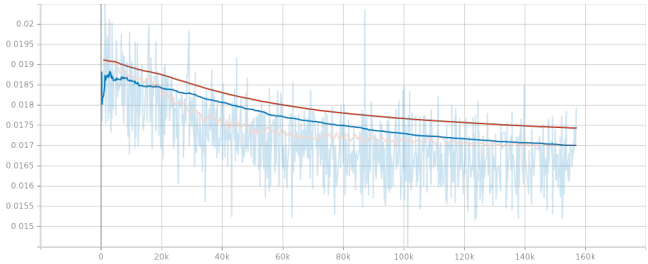

Fig. 3: Training loss curve after smoothing

Figure 4 displays the model's learnt filters for each output channel after the first convolutional layer in our final epoch. Each pixel represents a parameter for an output channel, with the darker colours corresponding to higher parameter weights,

Fig. 4: Learnt filters from first convolutional layer

IX. Qualitative Results

Generally, the model made fairly good predictions, an example of which can be seen in Figure 5. Although the spread of the visual attention area in the predicted saliency is higher than that of the ground truth, the area of higher attention focuses on the surfer. As desired, the saliency values transition smoothly between pixels, giving us realistic outputs.

Despite good accuracy scores when evaluating the predicted saliency maps, we found some interesting failure cases. The shallow network suffers from a strong centre bias, which can be clearly seen in Figure 6. The ground truth has two distinct yellow spots where the faces in the input image are, but our model produced a saliency map which gives a strong prediction at the centre of the image. Even in Figure 5, one can see that there is a very tight area of high attention in the centre of the prediction. Another interesting observation of Figure 6, which can also be seen in many other outputs, is that the model predicts very wide areas of visual attention.

The final failure case we observed can be seen in Figure 7, which shows a complete failure in the prediction. Although you can vaguely see an area that conforms around the ground truth, there are two large, intense, yellow spots which are clearly different from the ground truth. Not only are these areas incorrectly predicted, but they are given the highest likelihood of visual attention. Furthermore, the pixel values do not smoothly transition into their neighbours or the rest of the map; a key requirement of saliency maps. Multiple runs showed the same input images presenting this problem in their predictions.

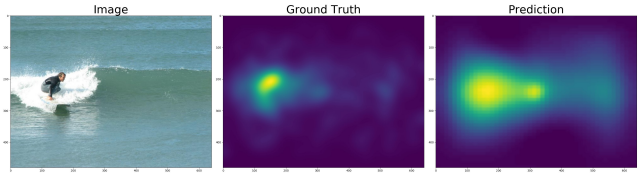

Fig. 5: Example of a good prediction

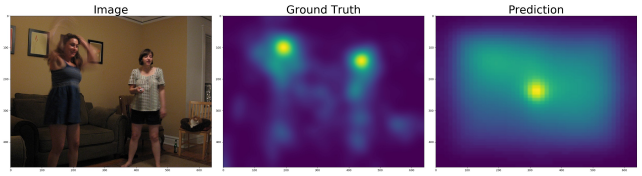

Fig. 6: Example of strong centre bias

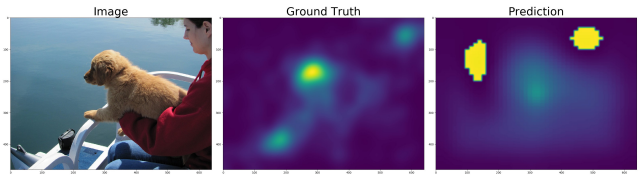

Fig. 7: Example of bad predictions

X. Improvements

We decided that to improve the accuracy of the model, we could train using a different loss function. The shallow

network suffers from a strong centre bias, so we believed choosing a new loss function would reduce the impact of this. Our intuition was that since L1 loss calculates the mean absolute error, the impact of the centre bias on the overall loss would be reduced. This would lead to the network better reducing loss from the actual errors between the prediction and ground truth, as well as making our model more confident in its predicted pixel values.

To implement L1 loss in our code, we simply define the criterion to be the L1 loss function from PyTorch. There are no other details that need changing in order for this to work correctly. Table IV shows the accuracy of our improved model compared to the previous replicated results. We saw a 6% increase in accuracy for the CC metric and a 4% increase for the AUC Shuffled metric compared to our replicated results. The accuracy of the AUC Borji metric was only 0.4% higher, so this could simply be variance in two runs of training the model, rather than an improvement from the loss function.

SALICON (val)	CC	AUC Shuffled	AUC Borji
Replicated Results	0.659	0.556	0.715
L1 Regularisation	0.70	0.577	0.718

TABLE IV: Comparison of replicated and improved results.

Figures 8 and 9 are both predictions from our improved model. Figure 8 is an example of a very good prediction. You can see that the shape of the saliency map closely resembles that of the ground truth, and the confidence of the prediction is high in the correct areas. Importantly, the centre bias is very small now. This can be seen clearly in Figure 9. This is the same input and ground truth as that in Figure 6 from earlier, which we showed has a large centre bias. Figure 9 still has a small centre bias, however the model now makes fairly strong predictions in the same areas as the ground truth. This shows that the centre bias is being mitigated, as we originally predicted.

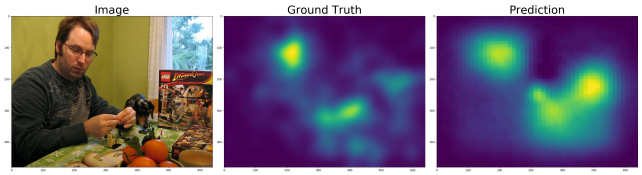

Fig. 8: Strong prediction example

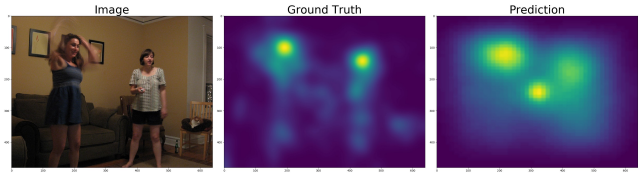

Fig. 9: L1 prediction of Fig. 6 image

Figure 10 shows the training and validation loss curves for our improved model training. One can see that that learning slows down much faster than the original implementation, as the loss trend levels off at around step 25,000 (epoch 159). Unfortunately, it appears the the model is over-fitting more than the original, although this is not by a large amount and doesn't appear to get worse as the training progresses.

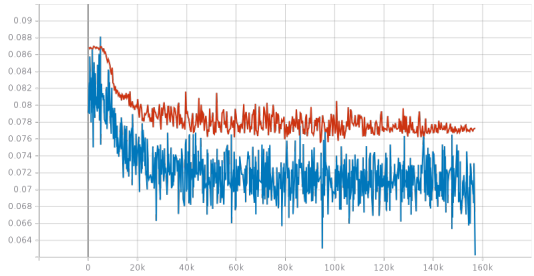

Fig. 10: L1 loss training loss curve

XI. Conclusion and Future Work

In this report we have successfully replicated the results from *Pan et al*'s paper, showing that handcrafted features can be replaced by data-driven methods. We also successfully displayed that by using the L1 loss function we can produce more accurate saliency map predictions. There is still a slight problem of centre bias and over-fitting in our model's predictions, however, using different a loss function such as KL divergence and Bhattacharya distance could provide better results, as these functions measure divergence and similarity between distributions, which would be better suited for saliency map prediction [9]. Additionally, alterations such as adding in a Learned priors module to more accurately assess centre bias [3], or replace our loss function with adversarial training which has been seen to provide accuracy improvements [4].

References

[1] Junting Pan, Kevin McGuinness, Elisa Sayrol, Noel O'Connor, and Xavier Giro-i-Nieto, "Shallow and Deep Convolutional Networks for Saliency Prediction" in IEEE CVF Computer Vision and Pattern Recognition (CVPR), 2016.

[2] Ming Jiang, Shengsheng Huang, Juanyong Duan, Qi Zhao, "SALICON: Saliency in Context", CVPR 2015

[3] Marcella Cornia, Lorenzo Baraldi, Giuseppe Serra, Rita Cucchiara, "SAM: Pushing the Limits of Saliency Prediction Model"; Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops, 2018, pp. 1890-1892

[4] J. Pan, C. Canton-Ferrer, K. McGuinness, N E. O'Connor, J. Torres, E. Sayrol, X. Giro-i-Nieto, "SalGAN: visual saliency prediction with adversarial networks" in Computer Vision and Image Understanding, 2018

[5] M. Cornia, S. Pini, L. Baraldi, and R. Cucchiara, "Automatic image cropping and selection using saliency: An application to historical manuscripts" in Digital Libraries and Multimedia Archives, 2018, pp. 169-179

[6] M. Cornia, L. Baraldi, G. Serra, and R. Cucchiara, "Visual Saliency for Image Captioning in New Multimedia Services" In ICME Workshops, 2017

[7] G. Underwood, T. Foulsham, E. van Loon, L. Humphreys, J. Bloyce, "Eye Movements During Scene Inspection: A Test of the Saliency Map Hypothesis" in European Journal of Cognative Psychology, 2006, 18:03, pp 321-342

[8] S. Ruder, "An overview of gradient descent optimization algorithms", 2017

[9] A. Bruckert, H R. Tavakoli, Z. Liu, M. Christie, O. Muer, "Deep Saliency Models: The Quest For The Loss Function" 2019

[10] PyTorch, https://pytorch.org/docs/stable/nn.html

[11] Lasagne, https://lasagne.readthedocs.io/en/latest/index.html

[12] *Pan et al* implementation, https://github.com/imatge-upc/saliency-2016-cvpr

[13] Python Software Foundation, https://docs.python.org/3/library/pickle.html