Artificial Neuron Responses using Transfer Learning

Tom Hodgkins
Department of Computer Science
Sheffield Hallam University
c0009574@my.shu.ac.uk

Abstract

In this paper I attempted to use transfer learning to train a model to replicate the activation's of the brain's visual cortex when viewing an image. I train a convolutional neural network for this task and use a modified cosine similarity as a loss function. I test and evaluate different hyper-parameters for the model and evaluate their performance on the given task. I found the neural network performed poorly on this task due to over-fitting. I think this was due to the limited size of the dataset.

1 Introduction and Implementation

I tried to produce responses from all regions of the brain using the provided dataset (Kay et al., 2008). I chose to replicate all the regions responses to reduce complexity in the system as different regions would not have to be filtered out. Furthermore, this meant that if successful a single model could be used to produce accurate responses for all regions. This is in contrast to region by region implementation where each region has one model and they are assembled to replicate all the region's responses.

Transfer learning is the method I am using to train my model. Transfer learning is the process of replicating one models output performance to another. This is very common in areas such as natural language processing as small language models can train in a semi-supervised fashion to replicate the performance of a large language model. In this case we cannot use a semi-supervised training method as the performance I am replicating is from a dataset (the brain) and not another model. This limits the data I have to use for training which limits transfer learning's ability to perform well. This is because transfer learning usually relies on massive amounts of data where labels are generated by the larger model.

The loss function for this task was very important to get right as it effects how well the model can align with the results. I chose to use the a modified cosine similarity between the dataset's responses and my models output to determine the loss. Cosine similarity outlines how similar two vectors are. Cosine similarity ranges from -1 to 1 with 1 meaning the two input vectors are exactly the same and 0 meaning the two vectors are orthogonal. As seen in my explanation the more similar a vector is to another the higher the cosine similarity. This is incompatible with gradient descent as it tries to descend to a minimum meaning a model trained with an unmodified cosine similarity loss function would converge towards vectors that a directly opposite to the dataset's responses. For this reason I created a custom loss function to flip and shift the original PyTorch cosine similarity function so that it would be compatible with gradient descent. This changed the range of the cosine similarity loss function to 0 to 2 as well as making 0 the value for vectors that are the exact same. This loss function was now compatible with gradient descent.

A way to measure the accuracy of the model is needed as well as the loss function. Without an appropriate accuracy measurement a model's performance cannot be properly evaluated. As you have probably worked out the cosine similarity can be used as a performance metric. I modified the original cosine similarity function so that it would output a percentage between 0% - 100% (50% being orthogonal vectors) representing how close the model's output vector was to the truth vector.

Now that the required functions were outlined a model architecture needed to be selected. I was unsure where to start so I copied the example convolution neural network from the PyTorch documentation (PyTorch, n.d.) as a starting point. I adjusted the input and output layers to be compatible with the dataset and task. Figure 1 shows the models architecture outline with the input images convolutional model and output artificial responses.

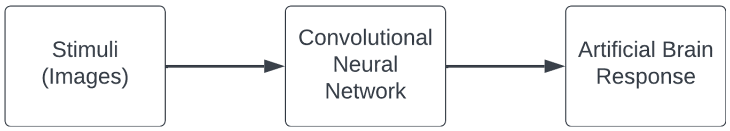


Figure 1: Overall Model Pipeline

2 Methodology

I tested a variety of different hyper parameters applied to the base model mentioned above. These parameters included changing the depth and width of the model drastically, changing the number of trainable convolution layers and linear layer trainable parameters, testing different optimizers, testing regularization techniques such as weight decay (Loshchilov & Hutter, 2019) and dropout (Srivastava et al., 2014) to try to reduce over fitting and reducing overall model size to try reduce over fitting. The subsections in the result's sections show the result graphs for the training and validation accuracy and the loss as well as an evaluation.

TotalConvolutionLayer(TrainableConvolutionParameters)-TotalLinearLayer(TrainableLinearParameters)-Optimizer-LearningRate-LossFunction-Epochs-Batchsize-FinalTrainingAccuracy-FinalValidationAccuracy-AdditionalParams-DateTimeSaved

Figure 2: Naming convention for the models

3 Results

3.1 Adjusting the Size of Linear Layers

Figure 3 and 4 show that adjusting the size of the linear layers impacts performance of the model. The models training accuracy seems to increase with larger linear models. However, all models seem to over fit as shown in Figure 4 b where the validation accuracy can be seen at around 50% for all models meaning the vectors produced by the model are orthogonal to the truth vectors in the validation set. This is a solid improvment but means the model will not be able to generalise to new input stimuli.

3.2 Adjusting the Size of Convolution Layers

Figure 5 and 6 show the results from 3 different amounts of feature maps and convolution layers with similar sizes of linear layers. The model with 22 feature maps performs the best getting 79.45% training accuracy. However, these models have the same over-fitting issues as the models in the previous section as their validation accuracy was not able to increase above 53%.

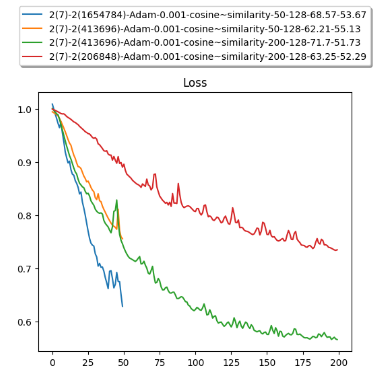
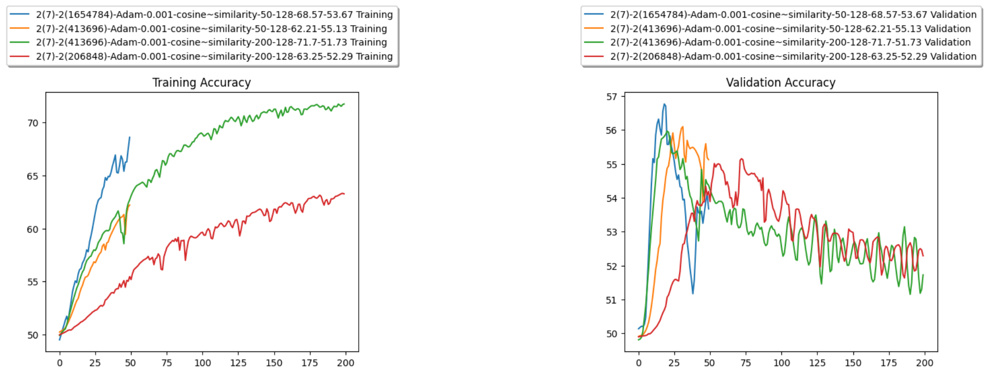


Figure 3



(a) (b)

Figure 4: Linear layer testing graphs

3.3 Width Versus Depth

This sections covers testing between a deep model and a wide model. Both models have similar amounts of trainable parameters. I only changed the linear layers in the model to make the model 'deep' or 'wide'. As seen in figure 7 and 8 the wide model (90.04%) performs drastically better than the deep model (67.15% and 53.99%) in training accuracy. However, both types of model still have over-fitting issues as their validation accuracy does not exceed 53%.

3.4 Optimizer Testing

I tested the Adam optimizer which I had been using for all test cases against the stochastic gradient descent optimizer. Their validation performance is similar but still shows over-fitted models. The reason I decided to run a 500 epoch version of stochastic gradient descent was due to the fact that it's 200 epoch model shows a nice upward trend in validation accuracy as shown in figure 10 b, however, this trend did not continue for the 500 epoch model and it over-fit as well. Figure 9 shows that the Adam optimizer has a faster convergence rate than stochastic gradient descent which lead to me continuing to use it in all other test cases.

3.5 Regularization

I tested using dropout and weight-decay. These are two common techniques used to try to prevent a model from over-fitting. Dropout gives a percentage chance (set between 0 - 1) that a weight won't effect the next layer. Weight decay adds the weights into the loss function meaning the weights

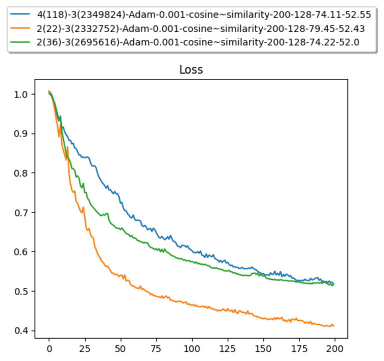
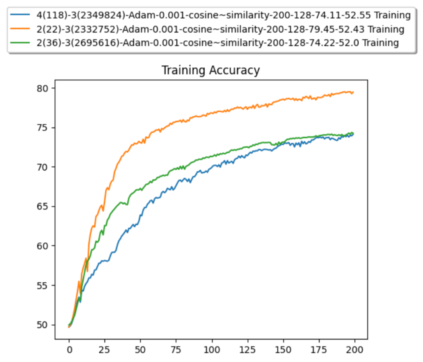
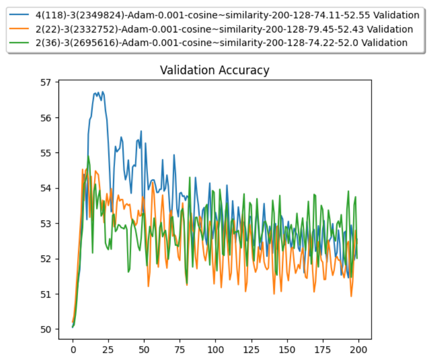


Figure 5



(a)



(b)

Figure 6: Convolutional layer testing graphs

now play a part in the gradient update. The weights are times by a weight decay constant before being applied leading them to ’decay’ over time. This can help prune the model of it’s unneeded complexity, however, it can still allow the model to over-fit or prevent the model from converging by ’decaying’ many weights too quickly.

I set the weight decay constant to 0.001 and my dropout rate slowly increased as the model got closer to output for 2 linear layers (50% chance then 70% chance).

Figures 11 and 12 show that the regularization techniques mentioned above did not help the model converge without over-fitting. All models still have an extremely low validation accuracy at around 53%.

4 Conclusion

In conclusion I think this paper was able to show results over a wide range of hyper parameters for the dataset given. However, the models performance was never high enough to generalise to new stimuli. This was due to an over-fitting problem. I think this was an effect of the small size of the dataset and complexity of replicating brain responses. If future work was to try improve on this more data should be collected to offer a greater chance of success.

5 References

[1] Kay, K. N., Naselaris, T., Prenger, R. J., & Gallant, J. L. (2008). Identifying natural images from human brain activity. Nature, 452(7185), 352–355. `https://doi.org/10.1038/nature06713`

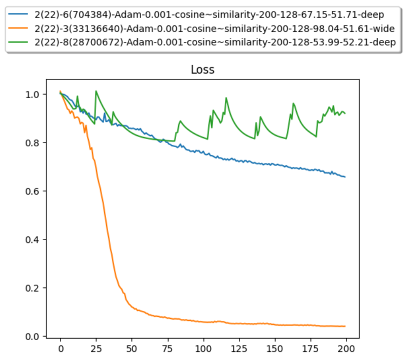
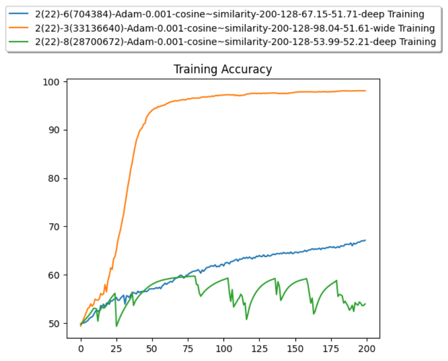
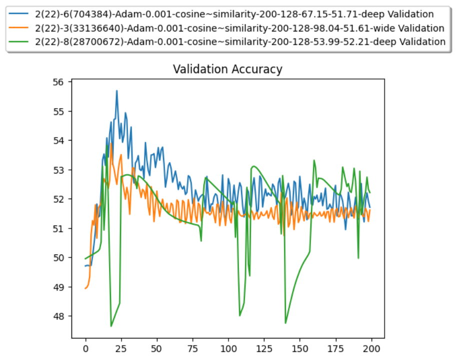


Figure 7



(a)



(b)

Figure 8: Width and Depth model graphs

[2] PyTorch. (n.d.). Training a Classifier — PyTorch Tutorials 1.5.0 documentation. Pytorch.org. Retrieved December 20, 2023, from `https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html`

[3] Srivastava, N., Hinton, G., Krizhevsky, A., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research, 15, 1929–1958. `https://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf`

[4] Loshchilov, I., & Hutter, F. (2019). DECOUPLED WEIGHT DECAY REGULARIZATION. `https://arxiv.org/pdf/1711.05101.pdf`

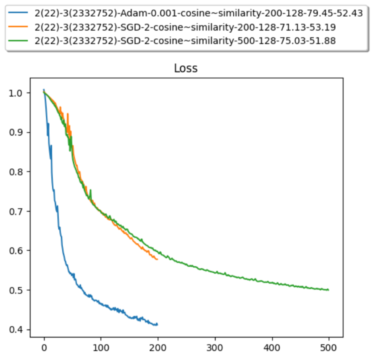
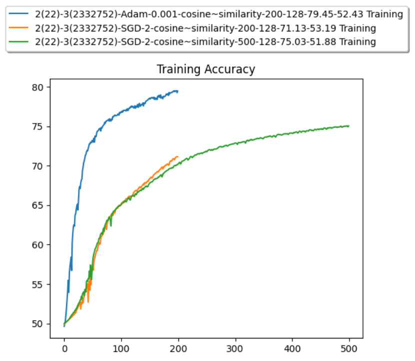
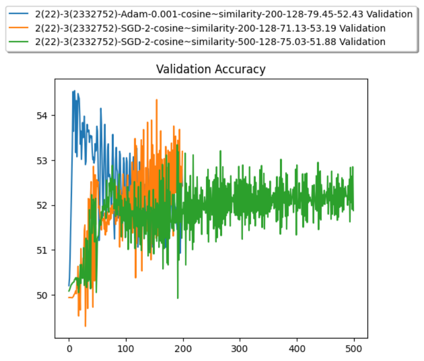


Figure 9



(a)



(b)

Figure 10: Optimizer testing graphs

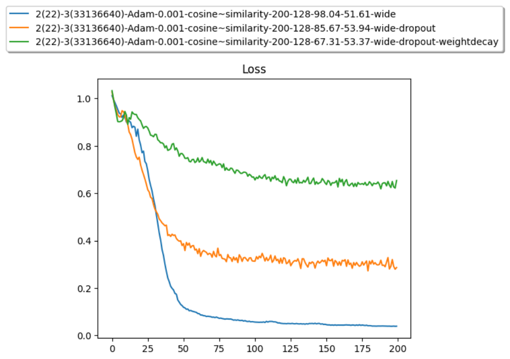
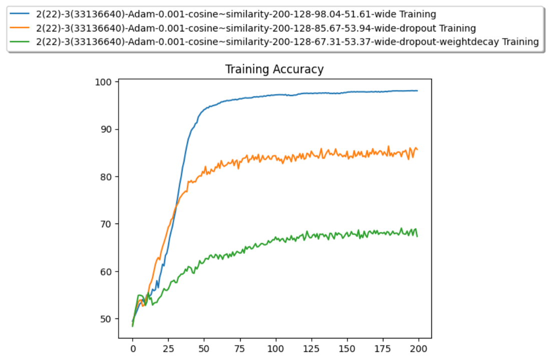
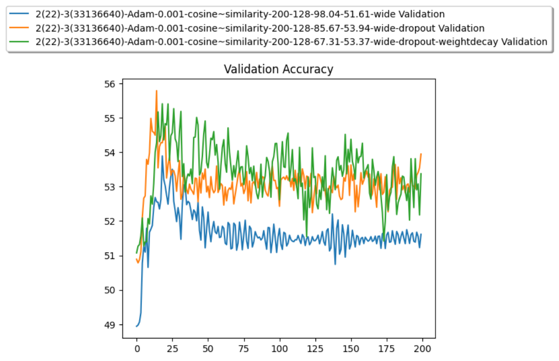


Figure 11



(a)



(b)

Figure 12: Dropout and Weight Decay model Graphs