Milestone #4 Report

By John Danekind and Daniel Hatakayama

Problem Space

Within the last few decades, with constant advancements in hardware, people are now able to utilize deep neural networks more easily than ever before. Because of this, we have seen a wide adoption across numerous industries and disciplines. More specifically, with the advancements in computer vision in recent years, people have been able to use it in a multitude of different ways and across various sectors such as healthcare, law enforcement, and public services, to solve problems. In the medical field, computer vision models such as convolutional neural networks can be used to assist radiologists and other medical professionals in making diagnoses. These algorithms help medical professionals by making the diagnosis quicker and more efficient, which can ultimately help save lives. Despite these benefits, convolutional neural networks are often viewed as a “black box” and people are unaware and unable to see how the model arrives at its final decision/output. However, in certain use cases like medical imaging, the final output/decision of the model has very real world effects. Because of this, transparency should be a top priority when producing these models. For our project, we implemented a convolutional neural network to classify brain tumors in medical images. While several projects like this have been done in the past, very few offer an insight into how the model actually works. With transparency as a priority, we implemented Gradient-weighted Class Activation Mapping (Grad-Cam), a technique developed by researchers at Georgia Tech in 2016 on top of our CNN.

Data

The dataset we used was from the website Kaggle. The brain tumor dataset contains separate CSV files for training and testing. In total, the dataset contains 4600 images with 2513 images belonging to the brain tumor class and 2087 belonging to the healthy class. In addition, the images are in two separate folders, “Brain Tumor” and “Healthy.” The key features of the dataset include:

1. **ID**: Unique identifier for each image.
2. **Image**: Filename or path to the image file.
3. **Format**: File format of the image (e.g., JPEG, PNG).
4. **Mode**: Color mode of the image (e.g., RGB, grayscale).
5. **Shape**: Dimensions of the image (e.g., width x height).

These features are important for our data pipeline to our preprocessing and model building steps. After reading in both folders, we split them into separate dataframes. After that, we split the data into three distinct sets: a training set, a testing set, and a validation set that is a small subset of the training set that is used to check for overfitting in training. We split the data 70% and 30% for training and testing. Then, we took 15% from the training and added that to the validation set. The dimensions of the data are:

- Training set is of shape (2932, 2)
- Testing set is of shape (1380, 2)
- Validation set is of shape (483, 2)

After all of this, we were ready to preprocess the images and create our model.

Approach

Before we built the model we did some preprocessing on the data. For each dataset, image preprocessing was conducted using TensorFlow's ImageDataGenerator to normalize, resize, and format the images for training, validation, and testing. Each image's pixel values were normalized from [0,255] to [0,1] where 0 is all black and 1 is all white. This reduces numerical instability when training the model. We used the flow_from_dataframe method to load images from file paths in the dataset DataFrame. All images in the dataframe were resized to 256x256 pixels and loaded in RGB format. The labels were one-hot encoded and we used a batch size of 64 for each different dataset. In the training data, shuffling was introduced to enable randomness in order to prevent overfitting. In the training and validation sets however, we did not use shuffling to ensure consistency in the evaluation. This preprocessing pipeline ensured efficient data handling, consistency across datasets, and optimal input preparation for the model.

After all of the data preprocessing, we built our Convolutional Neural Network using TensorFlow's Sequential API. Our model begins with a 2D convolutional layer consisting of 32 filters, and 3x3 kernel, this is followed by a ReLU activation function to introduce non-linearity. This is paired with a max pooling layer to reduce the spatial dimensions of the feature maps while retaining critical features. Our network then continues with a second convolutional layer of 64 filters and another max pooling layer, and a third convolutional layer with 128 filters. The increasing number of filters is intended to progressively capture more complex spatial hierarchies in the images. We use max pooling after each convolutional layer to further down-sample the feature maps.

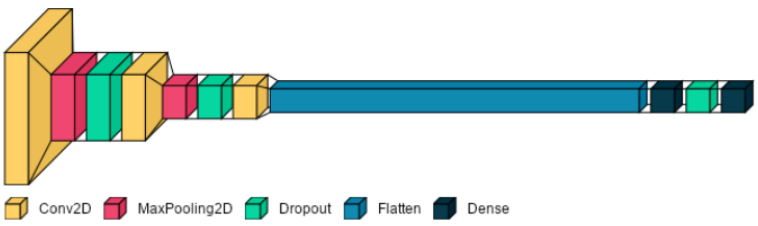

Fig 1. Visual representation of CNN structure

After the convolutional layers, our model then flattens the feature maps into a one-dimensional vector, which is then passed to a fully connected layer of 128 units and a ReLu activation

function. We also include a dropout layer at this stage in the model with a rate of 0.5 to prevent overfitting. This is achieved by randomly deactivating neurons during training. Finally, our output layer uses a softmax activation function with a number of units corresponding to the total number of classes in the training dataset which enables the model to perform multi-class classification by outputting a probability distribution over the possible classes.

Understanding what a convolutional neural network is "looking at" during its decision-making process is critical for interpreting and improving model performance. The motivation behind explainable machine learning is to demystify deep learning models, which are often considered "black boxes."

To achieve this, we use Gradient-weighted Class Activation Mapping (Grad-CAM), a powerful visualization technique that helps bridge the gap between model predictions and human interpretability.

Modern CNN visualization methods generate **heatmaps**, which highlight the regions of an image that are important for the model's decision.

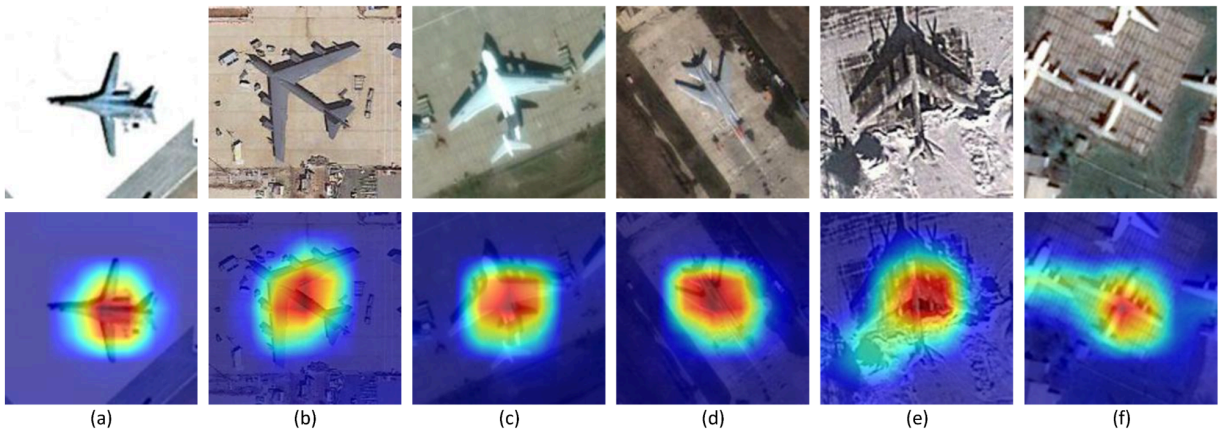

Fig 2. Heatmaps demonstrating the “attention” of a model trained on aircraft recognition. [src]

Grad-CAM improves on basic heatmaps by producing *class-specific visualizations*–it shows which parts of the image were most influential for a particular prediction, making it an invaluable tool for debugging and explaining CNN behavior. Grad-CAM overcomes limitations of earlier methods such as CAM (Class Activation Mapping), which required customized architecture to extract heatmap data.

At a high level, Grad-CAM computes the gradients of the target class score with respect to the feature maps in the last convolutional layer to identify important regions. These gradients are averaged to calculate weights, which are then used to emphasize the most relevant feature maps. The weighted maps are summed, passed through ReLU to focus on positive contributions, and resized to overlay on the input image as a heatmap.

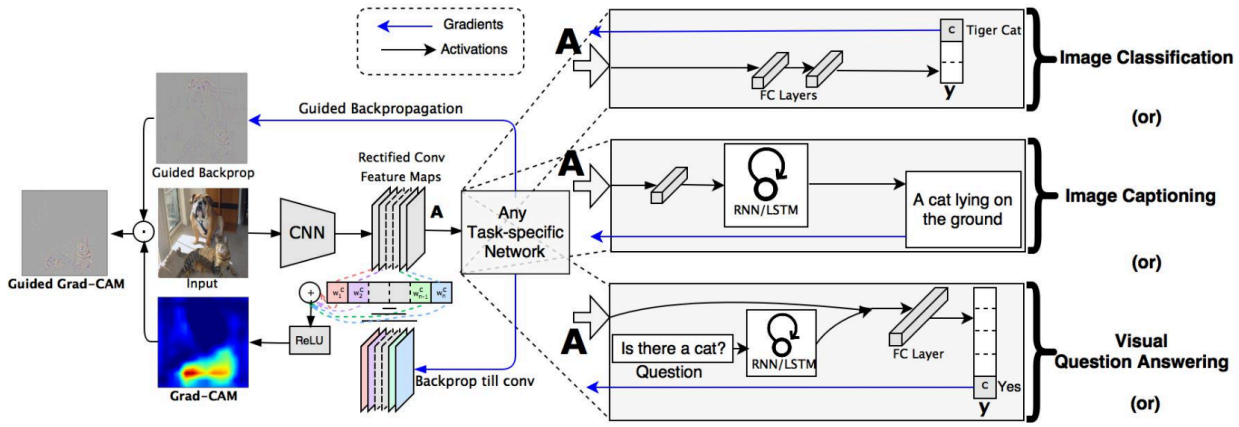

Fig 3. Data flow architecture overview of Grad-CAM. [src]

By revealing spatially localized activations tied to specific predictions, Grad-CAM allows for assessment of whether the model's reasoning aligns with human intuition and domain knowledge.

Our implementation and application of Grad-CAM utilized TensorFlow and Keras to provide visual interpretability of model predictions. Beginning with a preprocessed input image, we first generate a heatmap by computing the gradients of the predicted class score with respect to the feature maps of the last convolutional layer.

Using TensorFlow's GradientTape, we calculate the gradients of the predicted class score with respect to these feature maps, capturing how each feature contributes to the class prediction. These gradients represent the sensitivity of the class score to changes in the feature maps, providing the foundation for generating the heatmap.

```python
# Function to make Grad-CAM heatmap
def make_gradcam_heatmap(img_array, model, last_conv_layer_name, pred_index=None):
    # Get the last convolutional layer
    last_conv_layer = model.get_layer(last_conv_layer_name)
    last_conv_layer_model = keras.Model(model.inputs, last_conv_layer.output)

    # Get the classifier part of the model
    classifier_input = keras.Input(shape=last_conv_layer.output.shape[1:])
    x = classifier_input
    for layer in model.layers[model.layers.index(last_conv_layer)+1:]:
        x = layer(x)
    classifier_model = keras.Model(classifier_input, x)

    # Compute the gradient of the top predicted class for our input image
    with tf.GradientTape() as tape:
        last_conv_layer_output = last_conv_layer_model(img_array)
        tape.watch(last_conv_layer_output)
        preds = classifier_model(last_conv_layer_output)
        if pred_index is None:
            pred_index = tf.argmax(preds[0])
        class_channel = preds[:, pred_index]

    grads = tape.gradient(class_channel, last_conv_layer_output)
    pooled_grads = tf.reduce_mean(grads, axis=(0, 1, 2))
    last_conv_layer_output = last_conv_layer_output[0]
    heatmap = last_conv_layer_output @ pooled_grads[..., tf.newaxis]
    heatmap = tf.squeeze(heatmap)
    heatmap = tf.maximum(heatmap, 0) / tf.math.reduce_max(heatmap)
    return heatmap.numpy()
```

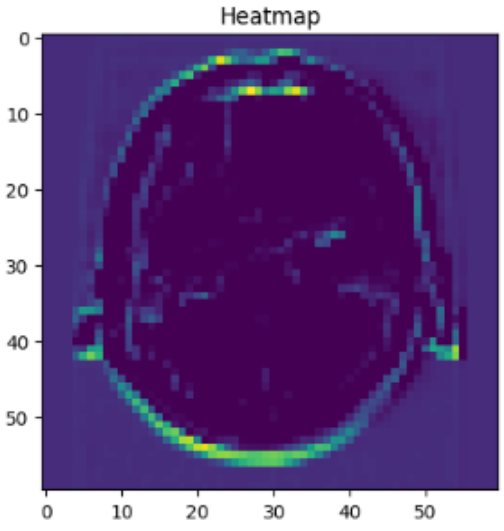

Fig 4. Code and generated Grad-CAM heatmap

These gradients are globally averaged using TensorFlow operations to obtain importance weights for each feature map channel, which are then used to weight and sum the feature

maps. The resulting heatmap is normalized and resized with OpenCV to highlight regions most relevant to the prediction.

Finally, we overlay these heatmaps on the original images using NumPy and Matplotlib for visualization.

Results

When training the model, we wanted to avoid the model overfitting to the training data. So, we used the validation set as a score after each iteration. We then made two plots: one showing the accuracy on the training vs the validation sets as a function of the number of epochs, and the other training vs validation loss as a function of the number of epochs.

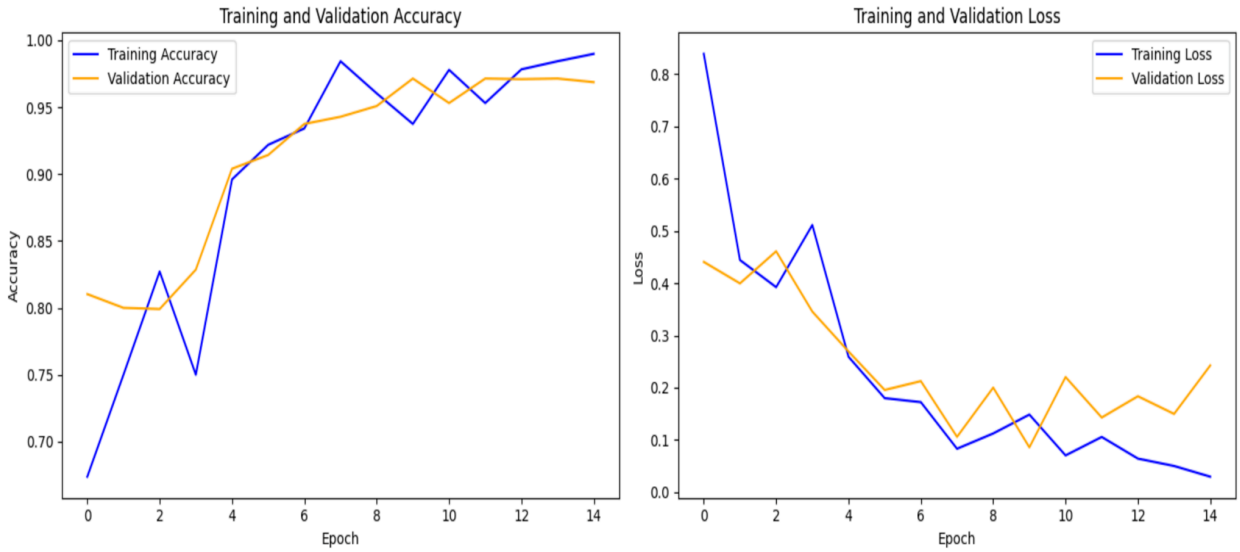

Fig 4. Plot of accuracy and loss for training and validation

As you can see, the overall accuracy of the training and validation is increasing with respect to the number of epochs. In addition, the loss is decreasing for both training and validation as we increase the number of epochs. This shows us that the model is capturing and learning from the data correctly. While there are some slight oscillations in both plots that imply slight overfitting to the training data, the general trend shows that the model is “learning” correctly.

After training the model over 15 epochs, we tested the model's accuracy on the test dataset using TensorFlow’s built-in evaluate function. We ran the model on all 22 batches of the testing data and at the end of each iteration, it would take the model's accuracy and subtract the loss. After all iterations were complete, we had an accuracy score of around 96% on average. To see how the model performed with respect to each class, we made a confusion matrix.

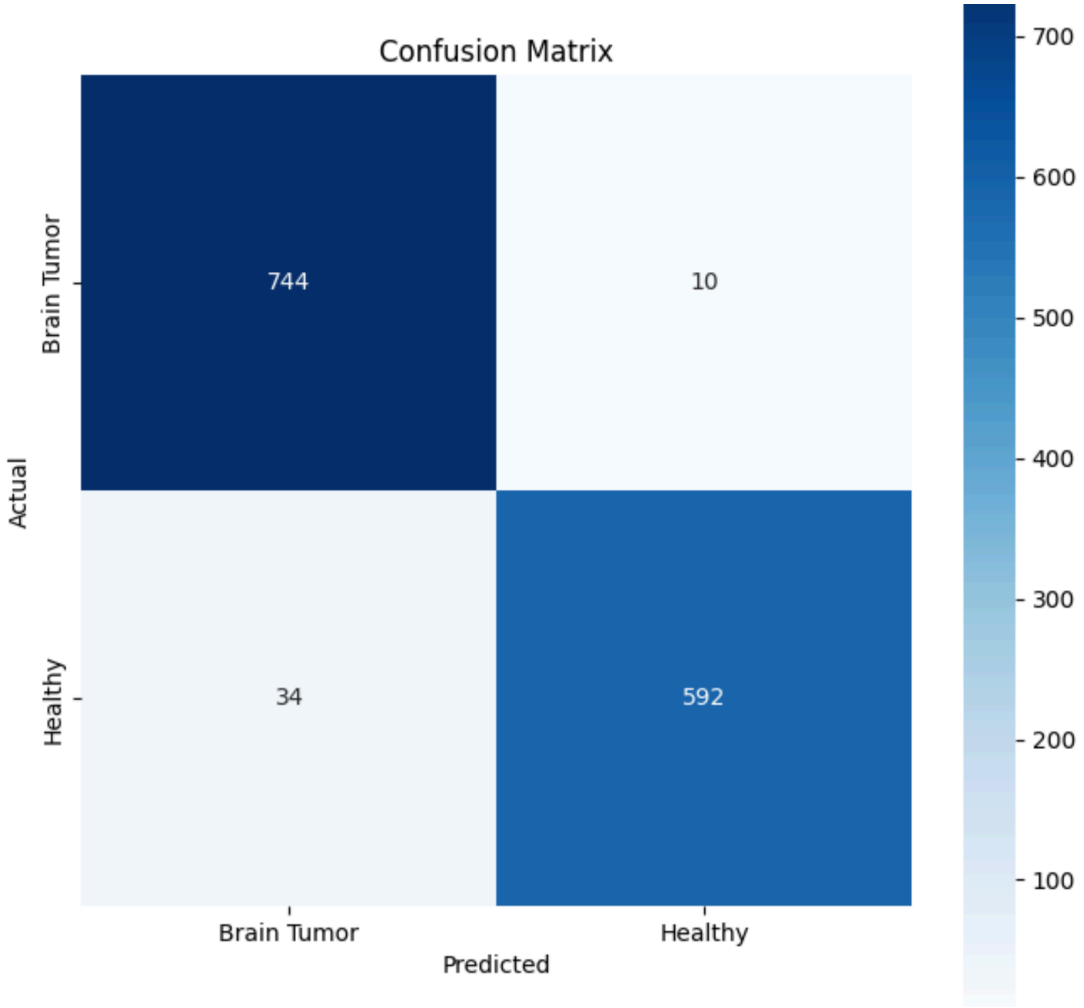

Fig 5. Confusion matrix for each class

Since there was a slight class imbalance where there were more images of brain tumors than healthy brains, we calculated accuracy from the confusion matrix in terms of F1 score. In this particular instance, we have an F1 score of approximately 97%. This shows that our model is balanced in terms of both precision and recall and it shows that we have very few false positives.

Despite our model performing very well in terms of accuracy and F1 score, our goal of the project was to investigate how the model ultimately came to its conclusion. After building the model, training the model, and calculating the models overall performance, we implemented Grad-Cam and displayed the original image, the image heatmap, and a superimposed image with the heatmap for comparison.

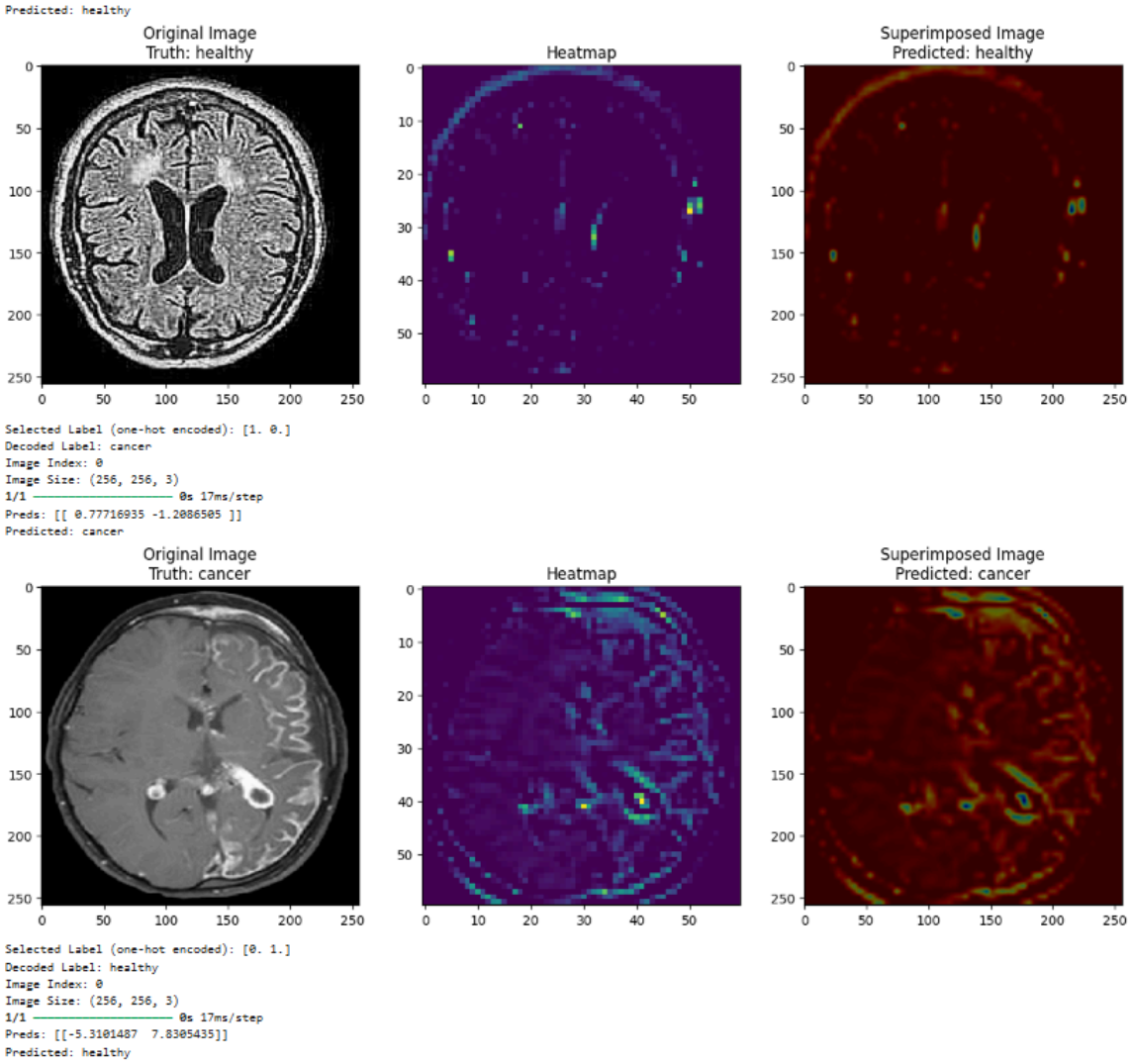

Fig 6. Heatmap that makes sense with clinical features

In certain instances, like the figure above, the pixels highlighted in the heatmap make sense. However, with other images, features of the image informing the models final decision were irrelevant clinical features. In the figure below, the skull shape informed the model's final decision.

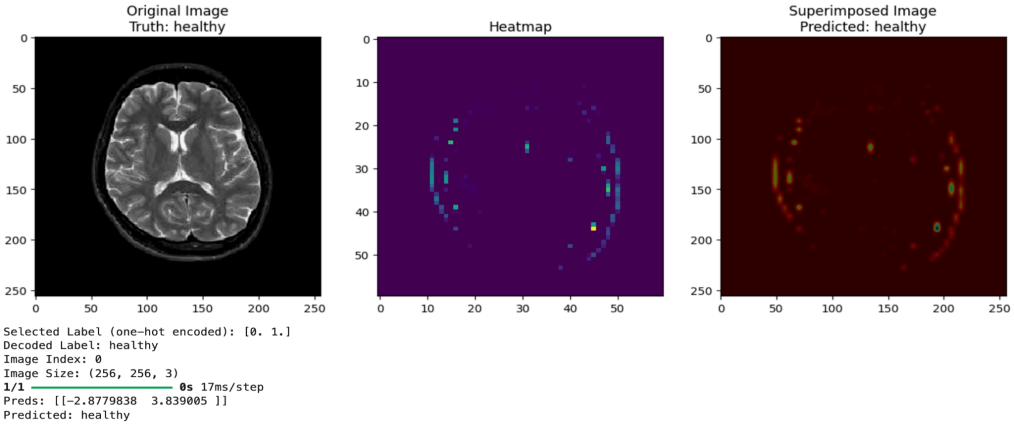

Fig 7. Heatmap with highlighting not relevant to clinical features

This shows us that despite a model's high accuracy, the way in which it outputs its final decision can be flawed with respect to the problem space. Grad-Cam illustrates this point.

Discussion

The results from our implementation of a convolutional neural network (CNN) for brain tumor classification provide valuable insights into both the capabilities and limitations of modern deep learning models. The model achieved an impressive average accuracy of 96% and an F1 score of 97%, which highlights how these models have impressive potential in aiding medical professionals in their diagnosis. However, our use of Gradient-weighted Class Activation Mapping (Grad-CAM) revealed critical areas in which the model relied on unintended features, such as skull shape, rather than directly relevant tumor-related characteristics. This further emphasizes the importance of interpretability in AI systems, especially in domains where real-world consequences are significant.

Looking at 2024 and beyond, AI will continue to be integrated into society more and more. The need for transparency and trust in these systems becomes increasingly important. In healthcare, where decisions have life-altering implications, understanding how and why a model arrives at a particular outcome is essential. Without transparency, there is a risk of relying on biased, unreliable, or unintended associations, which could ultimately lead to inaccurate diagnoses with potentially harmful consequences.

These findings show the importance of developing models that are not only accurate but also interpretable and robust against unintended correlations. Moving forward, we feel that the field must prioritize methods that ensure both high performance and explainability, enabling clinicians to gain insight into how AI is functioning and where its limitations lie. Future work should focus on enhancing interpretability by refining existing techniques as well as developing new techniques that can be tailored to specific domains.

Ultimately, our study highlights the importance of transparency in AI systems as they are adopted more widely across various domains. It has shown that while these models are impressive in their performance, a human expert is still essential when it comes to making important and life altering decisions. By ensuring that AI models are interpretable, we can foster trust and accountability, paving the way for responsible and ethical use of artificial intelligence in our society.