Analyzing deep learning models using saliency maps

Winston (Yuntao) Wu[*]

Abstract

In this project, I compare four different models (fully connected NN, shallow CNN, ResNet and EfficientNet) based on their performance on CIFAR-10 dataset. Saliency maps are used to check which pixels and features contribute the most to the final prediction. The CNN models no doubt performs better than the FC model. EfficientNet, a recent deep learning model for image classification, has the highest accuracy (74.36% under my configuration) while keeping a comparably smaller size. However, from the saliency map, there are still some irrelavent features learned, which might have an impact on the trustworthiness of the model.

1 Introduction

In this project, I implement the saliency maps method discussed in Christoph's *Interpretable machine learning* [1], which is an interesting topic I found when I wrote my reading report. I train a fully connected neural network, a convolutional neural network, a ResNet [3] and an EfficientNet [4] on the CIFAR-10 dataset [2]. Then I apply the saliency maps to compare these four models, checking whether or not these models are learning from convincing features of image to make their predictions.

1.1 Code

The code for the final project can be found at

`https://github.com/yuntaowu2000/400m-final-project`

There are five Jupyter notebook files in this repository. The file "Compare.ipynb" includes the comparison in saliency maps of the four models I use. The other four files are used for training the models. Each of the four training files includes all essential code it needs for running.

1.2 Implementation Environment

The linear, shallow CNN, and ResNet models are trained on my personal computer. It uses GTX1050 mobile (2GB) + 16GB RAM for CUDA support. Pytorch 1.11 + cu113 and torchvision 0.12 + cu113 are used for building and training the models. PIL and matplotlib are used for image visualizations.

Because the EfficientNet is taking too long to train on my personal computer, I switch to Google Colab for it.

2 Dataset and Data Preprocessing

CIFAR-10 is used for this project because of its popularity and convenience [2]. The dataset is small in size, but has a large amount of training and testing samples.

I follow a similar approach as the provided code in HW4 to preprocess the dataset. The dataset is loaded in $N \times C \times H \times W$ shape, where N is the number of images, C is the channel size, H is the

[*]Student number: 53625018

height and W is the width of the image. The mean of the training set is subtracted from the training and testing set. However, I also keep a separate version of training and testing sets where the mean is not subtracted for data visualization. The text labels (class names) for the images are also extracted from the batches.meta file. I use the entire 50,000 training images for training and 10,000 testing images for validation, with batch size 16. The goal is to find the best model that could generalize well at the first place through validation. The actual test on performance is done using saliency maps.

3 Models

3.1 Common Configurations

In all four models, I use cross entropy loss and stochastic gradient descent with learning rate 0.01. The training script is the same as in HW4 with 500 maximum epochs. The training for fully connected NN and shallow CNN stops if there is no improvement in the last 5 epochs so that there could be more iterations to improve. The training for ResNet and EfficientNet stops if there is no improvement in the last 3 epochs. The output layers of ResNet and EfficientNet are replaced to output a tensor of size 10 instead of 1000 as predefined in pytorch's model zoo.

3.2 Fully Connected Neural Network

The related code is in the file "FC_model.ipynb". It consists of only fully connected layers (nn.Linear) and ReLu for non-linear activation.

The model can be defined by the following equations:

$$\text{Layer 1: } X_1 = Flatten(X), \text{ transforms } \mathbb{R}^{3\times32\times32} \text{ to } \mathbb{R}^{3072}$$

$$\text{Layer 2: } X_2 = ReLu\left(W^{[1]}X_1 + B^{[1]}\right), \text{ where } W^{[1]} \in \mathbb{R}^{1024\times3072}, B^{[1]} \in \mathbb{R}^{1024}$$

$$\text{Layer 3: } X_3 = ReLu\left(W^{[2]}X_2 + B^{[2]}\right), \text{ where } W^{[2]} \in \mathbb{R}^{1024\times1024}, B^{[2]} \in \mathbb{R}^{1024}$$

$$\text{Layer 4: } X_4 = ReLu\left(W^{[3]}X_3 + B^{[3]}\right), \text{ where } W^{[3]} \in \mathbb{R}^{512\times1024}, B^{[3]} \in \mathbb{R}^{512}$$

$$\text{Layer 5: } X_5 = ReLu\left(W^{[4]}X_4 + B^{[4]}\right), \text{ where } W^{[4]} \in \mathbb{R}^{256\times512}, B^{[4]} \in \mathbb{R}^{256}$$

$$\text{Layer 6: } X_6 = ReLu\left(W^{[5]}X_5 + B^{[5]}\right), \text{ where } W^{[5]} \in \mathbb{R}^{128\times256}, B^{[5]} \in \mathbb{R}^{128}$$

$$\text{Layer 7: } X_7 = ReLu\left(W^{[6]}X_6 + B^{[6]}\right), \text{ where } W^{[6]} \in \mathbb{R}^{32\times128}, B^{[6]} \in \mathbb{R}^{32}$$

$$\text{Layer 8: } y = W^{[7]}X_7 + B^{[7]}, \text{ where } W^{[7]} \in \mathbb{R}^{10\times32}, B^{[7]} \in \mathbb{R}^{10}$$

The final output $y \in \mathbb{R}^{10}$ is a list of 10 values, representing the likelihood of the image to be in each of the ten classes. No activation is used in the final layer to avoid the vanishing gradients problem of sigmoid or softmax. The prediction is based entirely on the maximum value in y, the output of the final layer.

3.3 Shallow Convolutional Neural Network

The related code is in the file "shallow_cnn_model.ipynb". It consists of two parts (Figure 1). Firstly, it applies three convolutional layers to extend the feature channel size. 3×3 filters are used for efficiency. Padding of 1 is used to preserve the spatial dimensions. A 2×2 max pooling is used to shrink the dimensions and select the strongest signal in each region. Then, the image is flattened and fed through 4 linear layers to the output in the same format as in 3.2.

3.4 ResNet

ResNet-18 in the torchvision model zoo without pre-trained weight is used for training my model in this project [3]. The related code is in the file "resnet_model.ipynb".

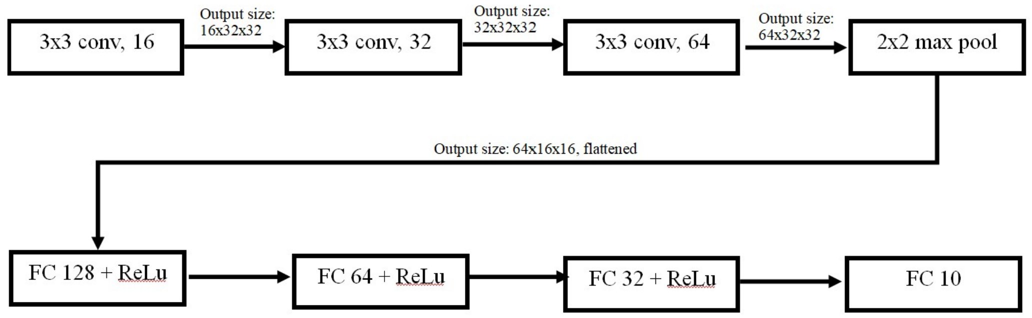

Figure 1: Shallow CNN model

Table 1: Comparison between 4 models

Models	FC NN on GTX1050	shallow CNN on GTX1050	ResNet-18 on GTX1050	EfficientNet-B0 on Colab
# Total params	4,889,834	**2,131,530**	11,181,642	4,020,358
Training time (second)	443.95	**407.51**	1453.37	4516.78
Training accuracy	61.66%	72.96%	77.13%	**84.15%**
Training loss	1.0857	0.7861	0.6464	**0.4547**
Validation accuracy	53.64%	65.54%	71.26%	**74.36%**
Validation loss	1.3733	1.0563	0.8514	**0.7434**

3.5 EfficientNet

EfficientNet-B0 in the torchvision model zoo without pre-trained weight is used for training my model in this project [4]. The related code is in the file "efficient_net.ipynb".

3.6 Comparison on Basic Metrics

The detailed comparison can be seen from Table 1 with the best value in each metric highlighted. My own implementation of the shallow CNN takes the least time to train and has the lowest number of total parameters, and it still outperforms the fully connected model, which doubles the number of parameters and takes longer to train. This shows the power of CNNs on image data.

ResNet-18 has the highest number of parameters, but it trains within a comparably short time, and gives a high training and validation accuracy. If we switch to ResNet-50 or ResNet-101, the results could have been better in terms of training and validation results.

Although EfficientNet-B0 takes the longest time to train, it performs the best in both training and validation [4]. It also uses considerably less parameters, only half of the parameters for ResNet-18, which is nice in terms of runtime memory and storage memory.

4 Saliency Maps

Saliency map is a pixel attribution method to interpret neural networks that highlights the pixels relevant for a classification [1]. It shows how each pixel of the input image contributes to the final prediction. The larger the gradient, the more the pixel contributes to the prediction.

I use the simplest gradient based method. Only the gradients towards the input image at each pixels are considered, while the gradients towards each parameter are ignored. The formula for the gradient calculation is $f(i,j) = \max_c \frac{dS}{dI}|_{I=(c,i,j)}$, where S is the maximum value in the output y (the score for the most likely class), $c \in [0,2]$ is the feature channel and $(i,j) \in [0,31] \times [0,31]$ is the pixel

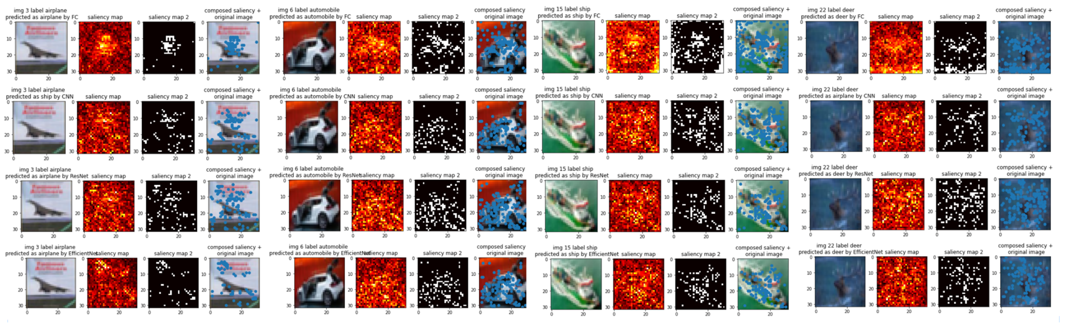

Figure 2: Saliency maps for 4 models on 4 samples. Clearer version can be seen in Compare.ipynb

of the input image at which the gradient is calculated. The gradient is then scaled to $[0, 1]$ using $g(i, j) = \frac{f(i,j) - \min_{(i,j)} f(i,j)}{\max_{(i,j)} f(i,j) - \min_{(i,j)} f(i,j)}$ (second image in each row in Figure 2). It is still hard to see what is happening with the preliminary saliency map calculated in this way, since the gradient differences of nearby pixels are fairly small and most parts of the full 32×32 region have some gradients. Therefore, I thresholded the gradients by $G(i, j) = \begin{cases} 1, & g(i, j) \geq 0.5 \\ 0, & g(i, j) < 0.5 \end{cases}$ to plot onto the original image (third & forth image in each row in Figure 2). This gives a better visualization.

The saliency maps created can be seen in Figure 2. The FC model is very likely to consider more pixels than other models, either grouped together or near the boundaries of the images. It also considers more backgrounds than other models, making it trustless. It might be predicting an airplane by the sky, a ship by the water and a deer by the snow. Models involving convolutional layers are much better in discovering features, as the gradients mostly flow back to the actual objects we want to classify. The EfficientNet performs the best among the models, with the fewest gradients flowing back to completely irrelevant pixels. This means that it is more likely to learn from the features of the objects to be classified and make decisions based on these highly relevant features. However, none of the models works well on image 3 (airplane), even the ones that give the correct prediction. As we can see, a large portion of the pixels with large gradient values are not on the airplane itself. Rather, significant gradients flow back to the red part in the top which could be some news titles that are irrelevant to the object in the image.

5 Discussion

This project applies a pixel attribution method, saliency map, to compare the performance of four different neural network models (two shallow models created by me and two state-of-the-art deep neural network models) on the CIFAR-10 dataset. With some processing, even the simplest vanilla gradient based saliency map can give us a good visualization on how a neural network learns from image data.

The saliency map provides an alternative to judge the performance of a neural network model than the basic metrics, such as accuracy and loss. It could also give a better visual explanation to non-experts, because it is easier for people to perceive visual than numerical data. Two distinct models may have close accuracy and loss, but the learned features may be different. We can apply saliency maps to check which model is better at learning correct features, and then choose this better model for real world applications, because being able to learn from correct features means that the model could potentially generalize better.

However, we should not judge the performance of a model based entirely on the saliency map. Avanti et al. point out that there is saturation problem when ReLu is used and when the activation is negative, and the vanilla gradient method based saliency map says that the neuron is not important in the case, because the gradient is 0 [5]. Also, Christoph says that pixel attribution method could be fragile and is yet to be improved [1].

Nevertheless, the method seems to work in my case and it gives some reasonable explanations on how the models make predictions. We shall wait and see more researches on how we can properly evaluate the pixel attribution method so that we can actually use this method to convince the public that our neural network model indeed works.

There are a few things that can be done as future work. Firstly, the stopping criteria is not chosen wisely. The loss is likely to fluctuate a lot before it finally converges to a global minimum. The current stopping criteria (no improvement in the last N runs) can definitely find a local minimum, but the model could potentially be improved by taking more epochs. A better choice could be checking the difference between the loss of the last two runs, if the difference is less than a threshold ($\text{loss}_t - \text{loss}_{t-1} < \text{threshold}$), we can stop. However, this could take a much longer time to train, and not work well with the time constraint of the project and the EfficientNet model. If we use up the entire 500 epochs, it would take around 20 hours. The second thing that is interesting to investigate is how the saliency map changes during each epoch. This could give us some hint on the learning process of the machine. However, this would require saving the models for each epoch and plotting the data using all the models we trained.

References

[1] Molnar, Christoph. *Interpretable machine learning. A Guide for Making Black Box Models Explainable*, 2019. https://christophm.github.io/interpretable-ml-book/.

[2] Krizhevsky, A. and Hinton, G. *Learning multiple layers of features from tiny images*, Technical Report, 2009.

[3] Kaiming, He, et al. *Deep Residual Learning for Image Recognition*, ArXiv:1512.03385 [Cs], Dec. 2015. arXiv.org, http://arxiv.org/abs/1512.03385.

[4] Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*, ICML 2019. Arxiv link: https://arxiv.org/abs/1905.11946.

[5] Shrikumar, Avanti, Peyton Greenside, and Anshul Kundaje. *Learning important features through propagating activation differences*, Proceedings of the 34th International Conference on Machine Learning-Volume 70. JMLR. org, 2017.