Adversarially-Trained Classifier

tags: **`cs 182`** **`deep learning`** **`computer vision`** **`ImageNet`** **`classifiers`**

Internal Purposes: the below includes bullet points to ensure that we are including all aspects of the rubric requirements in our order of choice.

1. Know your model’s purpose

The field of computer vision continuously calls for increased accuracy and performance o classifiers. Although people have been training models to increase model accuracy on datasets by the smallest percentage, the larger problem remains underexplored - building a robust generalizable classifier.

The goal of this project is to create models capable of classifying unseen data with unknown perturbations.

Startups from SenseTime and Hawk-Eye Innovations to other big corporations are all competing to build a robust model with a unique edge. Autonomous driving seems to be the most obvious application in today’s day and age, but its application spans across IOT technology and various other industries.

2. Understand your data

Imagenet is a classic visual database used to train models in computer vision. Tiny Imagenet is simply a subset within that database with smaller resolution images, and allows retail machine learning engineers to train their models much faster.

Our Tiny Imagenet dataset contains 200 classes, each consisting of 500 training images, 50 validation images, and 50 test images.

3. Choose the optimal toolkit

We considered TensorFlow with Keras and Pytorch for toolkit, and ultimately decided on TensorFlow with Keras since

1. it allows simple implementations of pretrained models with custom input size.
2. it contains a larger variety of pretrained models.
3. we are not implementing complicated training processes where Pytorch has the edge.
4. it allows a simpler way to understand model architecture that is readable and concise.

4. Building the core of the model

The major design choices in our project include

1. Augmenting input images with random horizontal flips, random cropping, and color jittering with common additive noises including Gaussian, Poisson, and salt-and-pepper.
2. Choosing MobileNet v2 as base model for having very high accuracy to number of parameters ratio among pretrained models and a reasonable model size for Tiny Imagenet.
3. Incorporating Neural Structured Learning’s Adversarial Regularization to improve model robustness by injecting adversarial loss in the training process.

We have considered implementing TRadeoff-inspired Adversarial DEfense via Surrogate-loss minimization (TRADES) [Zhang et al. 2019] but ultimately decided not to due to time contraint. It will be an interesting technique to explore in the future.

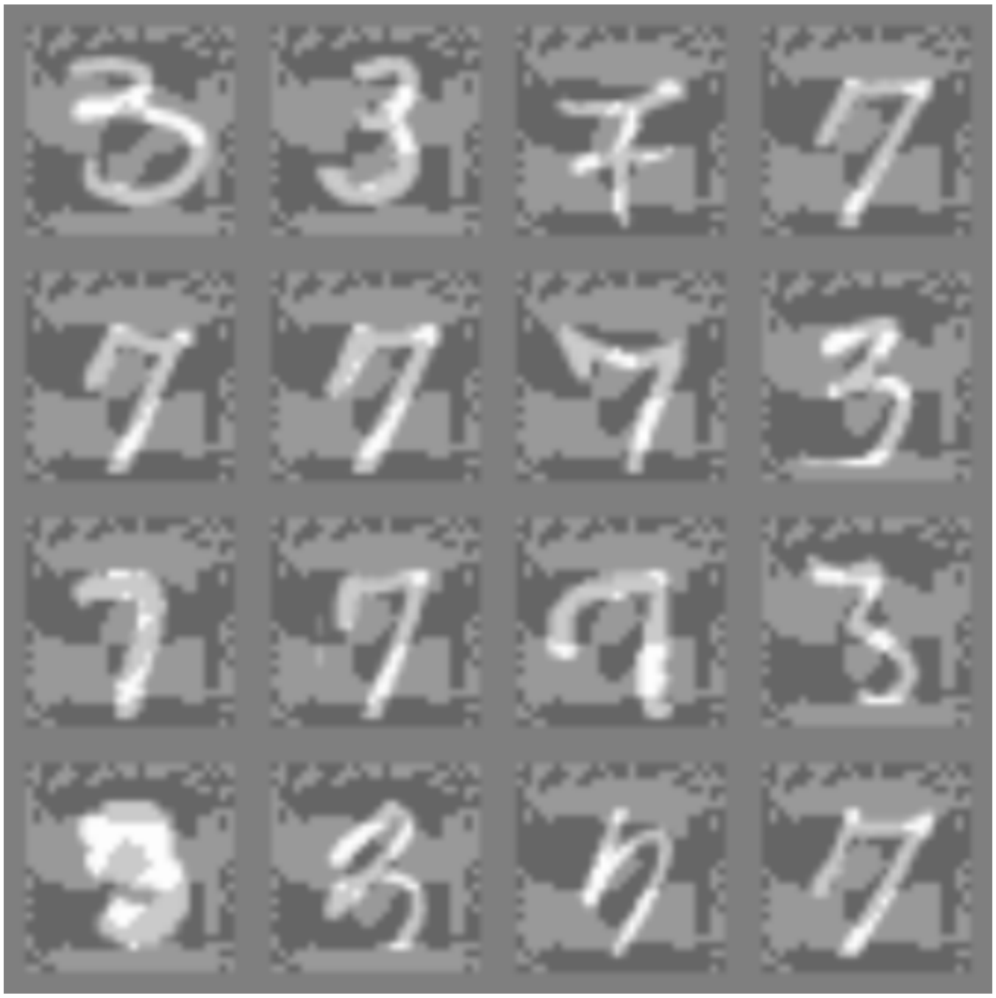
5. Combatting adversarial conditions

- Adversarial Regularization by adding perturbed data into training set. Even though recent neural networks for computer vision tasks has reached remarkable accuracy they are still extremely vulnerable to small, human-imperceptible perturbations. Previously

researchers have speculated that this is due to the nonlinear nature of the neural networks. However, authors of the paper, Explaining and Harnessing Adversarial Examples, this vulnerability is due to the linear nature of neural networks. For example, ReLU, LSTMs, have largely linear components. Even for nonlinear function, like sigmoid, the model will have most values in the linear regime, around (x=0). In this paper, the author provided a fast way to generate adversarial examples along with a adversarial training method, which is what we have implemented in our project.The author introduced, Fast Gradient Sign Method (FGSM), to efficiently generate adversarial examples. Calculate the gradient with respect to the input.

- And perturb input by: input = input + epsilon*sign(gradient)

- Below is an example, of FGSM applied to logistic regression model trained on MNIST threes and sevens.



- Perturbation was applied directly on sevens and inverted on threes to make the model misclassify as sevens. FGSM can be directly incorporated into the loss function, which also has an added regularization effect. In addition to this, the author suggested including advesarially perturbed inputs on top of the organic inputs and include adversarial loss as part of the objective function. With adversarial training the author achieved a better final model accuracy and increased the robustness of the model. The adversarial trained model had an error rate of 17.9% on adversarial examples against the 89.4% of the base model.

- This type of adversarial training suggests that it can translate well into our project where the test data can possibly be adversarially perturbed.

- To implement this model, one needs to install Neural Structured Learning (NSL).

- And use the nsl.keras.AdversarialRegularization wrapper.

```
base_adv_model = create_model('base_adv_model', img_dim, 2, len(classes))
adv_model = nsl.keras.AdversarialRegularization(base_adv_model, label_keys=['label
adv_model.compile(optimizer=keras.optimizers.SGD(learning_rate=lr, momentum=0.9),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

              metrics=['sparse_categorical_crossentropy', 'sparse_categorical_accu
                       'sparse_top_k_categorical_accuracy'])
```

Use adv_config to specify hyperparameters for the wrapper.

```
adv_config = nsl.configs.make_adv_reg_config(
    multiplier=0.2,
    adv_step_size=0.2,
    adv_grad_norm='infinity',
)
```

Multiplier is the regularization weight.
Adv_step_size is the step size to find the adversarial example. Corresponds to epsilon in the above equation.
adv_grad_norm specifies the tensor norm to normalize the gradients.

Training time procedure should be same as normal except dataset needs to be dictionaries instead of tuples.
This can be done simply by converting original dataset, train_data and val_data, to dictionaries.

```
IMAGE_INPUT_NAME = 'image'
LABEL_INPUT_NAME = 'label'

def convert_to_dictionaries(image, label):
  return {IMAGE_INPUT_NAME: image, LABEL_INPUT_NAME: label}



train_data_adv = train_data.map(convert_to_dictionaries)
val_data_adv = val_data.map(convert_to_dictionaries)
```

Instead of training on train_data and evaluating on val_data, run it with train_data_adv and val_data_adv.

6. Validating on Adversarial Examples

First create a reference model to create perturbed images,

```
reference_model = nsl.keras.AdversarialRegularization(
    base_model,
    label_keys=[LABEL_INPUT_NAME],
    adv_config=adv_config)
#optimizer was adam in documentation
reference_model.compile(optimizer=keras.optimizers.SGD(learning_rate=lr, momentum=
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
              metrics=['sparse_categorical_crossentropy', 'sparse_categorical_accu
                       'sparse_top_k_categorical_accuracy'])
```

Load the models and metrics you want to validate.

```
models_to_eval = {
    'base': base_model,
    'adv-regularized': adv_model.base_model
}
metrics = {
    name: tf.keras.metrics.SparseCategoricalAccuracy()
    for name in models_to_eval.keys()
}
```

The below code will validate both models.

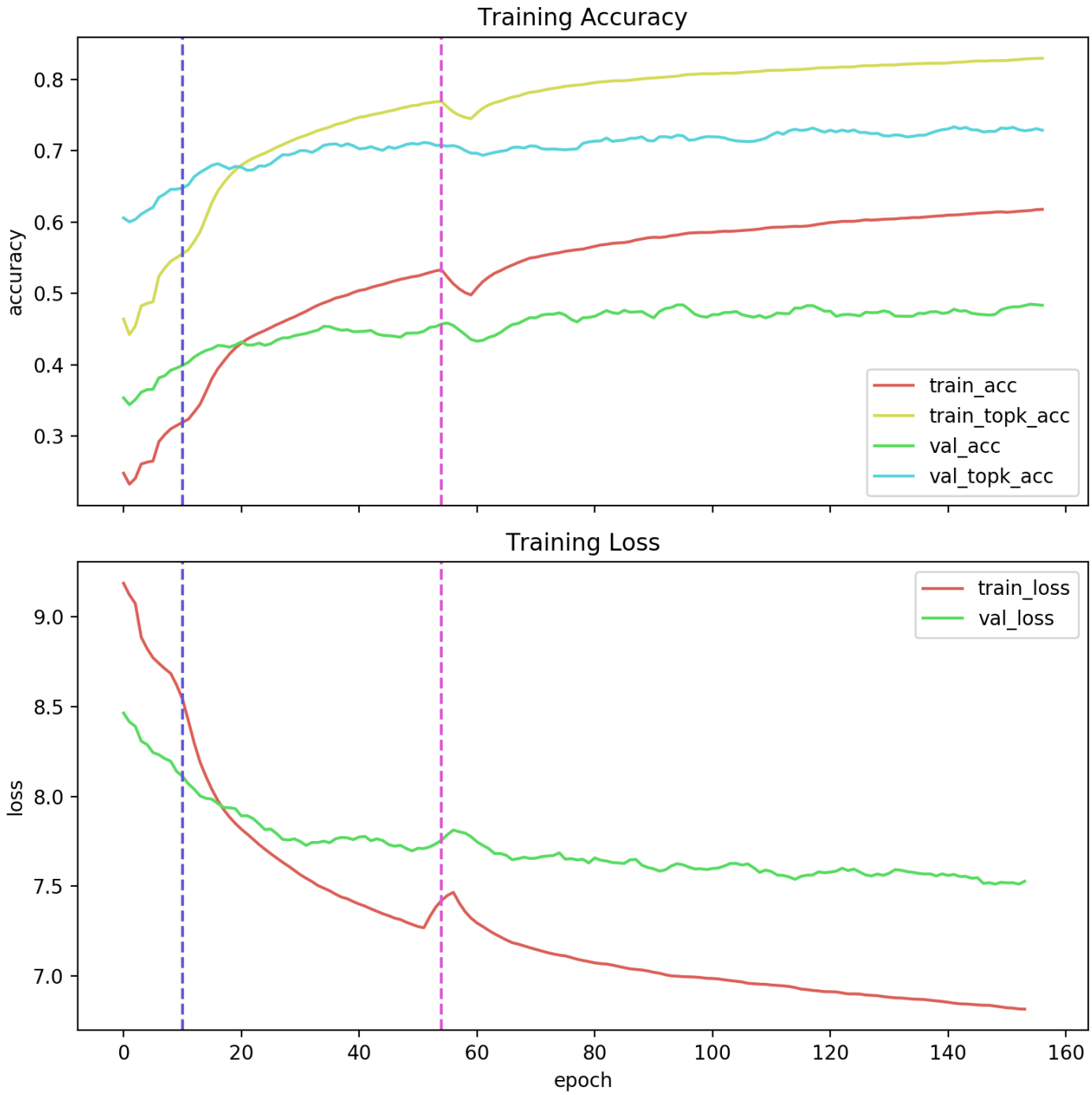
```
perturbed_images, labels, predictions = [], [], []

for batch in train_data_adv:
  #model used to be reference_model
  perturbed_batch = reference_model.perturb_on_batch(batch)
  # Clipping makes perturbed examples have the same range as regular ones.
  perturbed_batch[IMAGE_INPUT_NAME] = tf.clip_by_value(
      perturbed_batch[IMAGE_INPUT_NAME], 0.0, 1.0)
  y_true = perturbed_batch.pop(LABEL_INPUT_NAME)
  perturbed_images.append(perturbed_batch[IMAGE_INPUT_NAME].numpy())
  labels.append(y_true.numpy())
  predictions.append({})
  for name, model in models_to_eval.items():
    y_pred = model(perturbed_batch)
    metrics[name](y_true, y_pred)
    predictions[-1][name] = np.argmax(y_pred.numpy(), axis=-1)

for name, metric in metrics.items():
  print('%s model accuracy: %f' % (name, metric.result().numpy()))
```

```
base model accuracy: 0.500700
adv-regularized model accuracy: 0.960400
```

7. Review performance



The graph shows the curves on training and validation data, and the vertical lines indicate finetuning higher layers in the base model. It is apparent that even though we tried to combat overfitting as much as possible, by using data augmentation and regularizations, the model still overfits to the Tiny Imagenet dataset.

8. Fine tune your model

The major hyperparameters that we tuned are the base model for our network and learning rate. We used SGD in favor of ADAM due to previous experiences and papers suggesting that SGD is more suitable in CV tasks. Instead of deciding the number of epochs to finetune

different layers in advance, we move onto higher layers as training loss plateaus.

Network Choices

We have implemented our model with ResNet and its variants (ResNeXt, DenseNet), NASNet and NASNetMobile, EfficientNet B2 and B4 and B6, and lastly decided on MobileNet v2 as the base network. ResNet variants took longer to train with the large number of parameters / connections, and NASNet tend to overfit too much due to its strong expressive power on the small datasets that we have. We had initially expected NASnet to do well since it is the best performing model on the original imagenet.

EfficientNet was a very close contender, but due to its late addition to the TF pretrained models and the deadline of this project, we decided to go for MobileNet instead. It is fast to train which allows for frequent modifications to our project, and also expressive enough to learn our data resulting in more than 80% training accuracy.

Learning Rate

We have trained with learning rates of 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001 on both the classification layer and the pretrained model layers. We discovered that the usual training regime of starting high and decreasing on the classification layer works best, starting off with 0.01 and schedule it to step down to 0.001 when loss fluctuates. Then the last couple of layers of the pretrained model is finetuned with 0.001, and then decrease the learning rate to 0.0001 as we move up to higher layers.

Too high of a learning rate induces significant fluctuation when finetuning and too small fails to learn much, a theory that we understand but difficult to master in practice. Experimenting with learning rate is essential in these conditions for building a generalizable classifier. Batche size and layers in conjunction with learning rate play a big role in the performance.

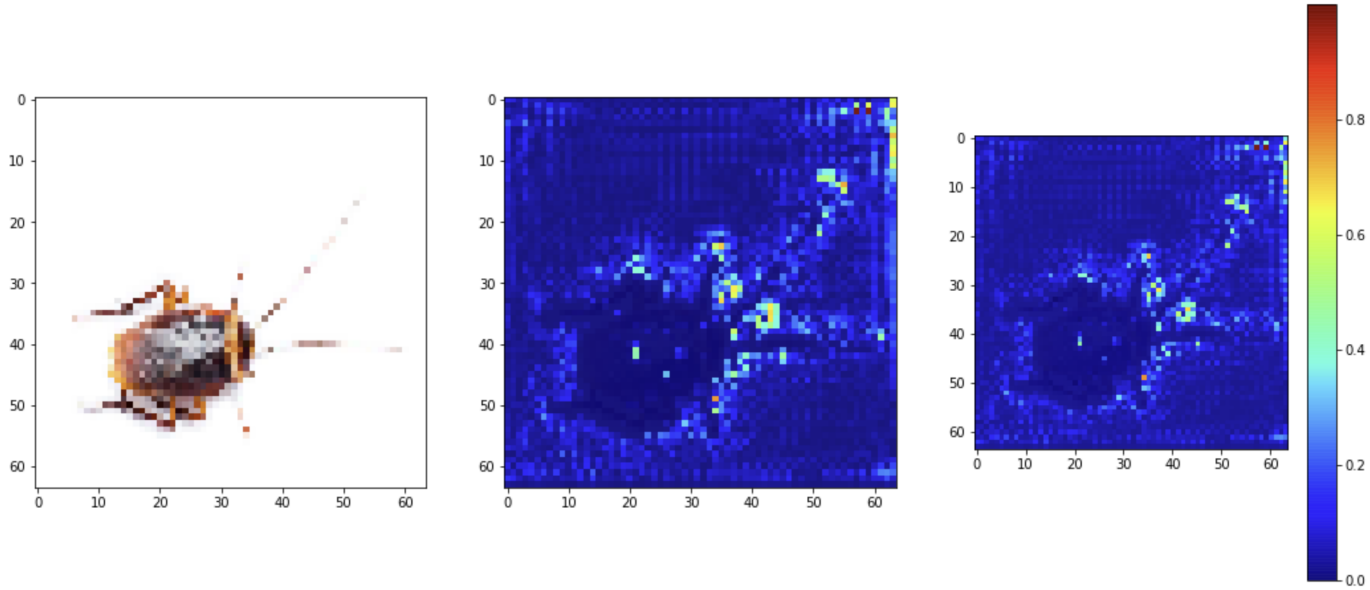
9. Explainable AI - Understand the method used

Saliency refers to unique features for a particular input in the context of visual processing. This method allows for emphasis on visually alluring locations on an image. There is a scalar quantity to help with the process of selecting certain locations to make processing more accurate.

There are many ways to visualize saliency such as guided back propagation. We simply changed the last layer to a linear activation to see which pixels had the biggest influence on classification decision. We used keras-vis module to do this.

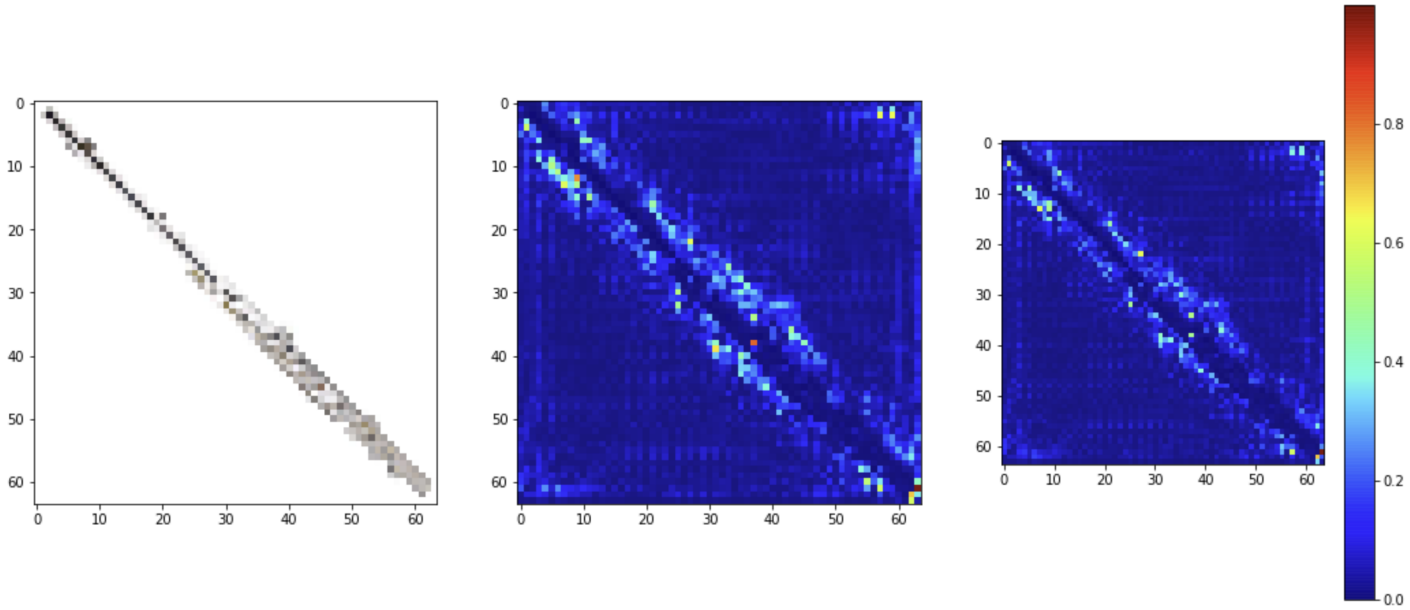
See the below examples with inputs of a cockroach, oboe, and syringe. We picked these images since they were classes that actually existed in Tiny Image Net. The two saliency maps shows the positive gradients and negative gradients respectively.

Our model was able to correctly predict the coachroach label. We can see the model was able to do this easily since linear activation maximizations in the last layer clearly shows the shape of cockroach.

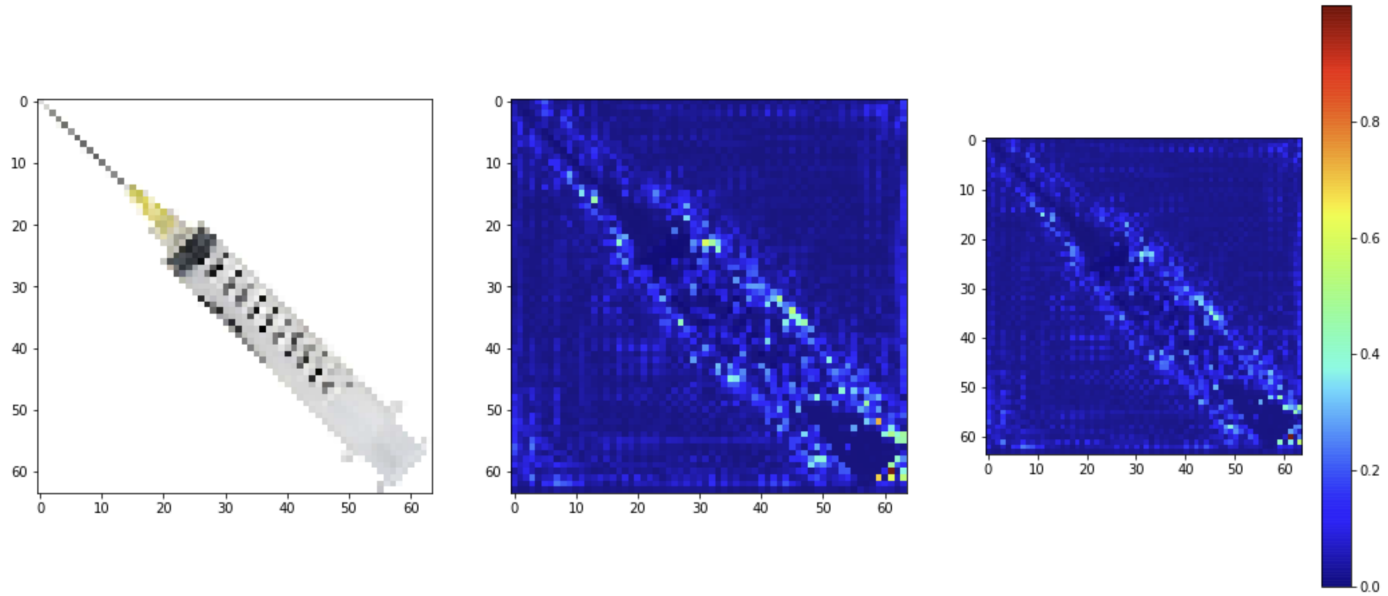


As seen, Cockroach whiskers are prevalent in this saliency map.

Our model did not classify the objects below correctly. This can be explained by the fact that Tiny Image Net has lower resolution, resulting in similar activations from objects with similar shapes. In the oboe and syringe example, the activations from these images are very similar so our model decided it was a toss between Oboe, Syringe, broom, and beer bottle. These were amongst the top 5 predictions.



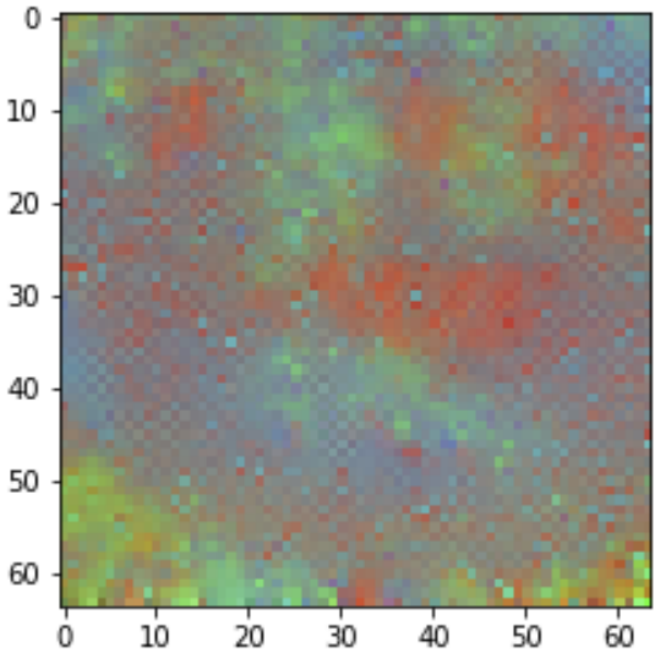
One of the top 5 classifications here was broom (n02906734). Which is not only reasonable, but also could be because of our random crops during training data augmentation, where we classified the handle portion frequently.



One of the misclassifications for the image was beer bottle (n02823428), which we can see from the activations as well.

We also decided to visualize some of the convolution layer outputs by stacking them together. This type of visualization resulted in images like the trippy art in VGG19, but unfortunately our's just looked random with very little patterns.

```
<matplotlib.image.AxesImage at 0x7f07eafe9f60>
```

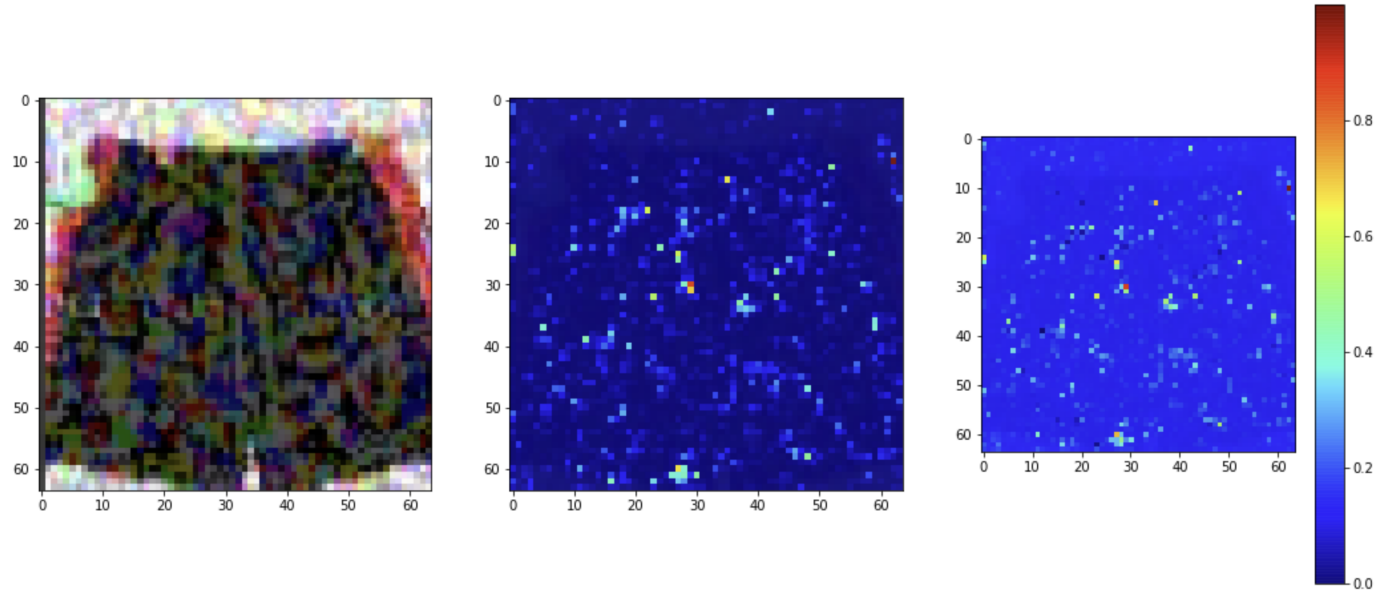


convolution layer output visualization for the class label of cockroaches

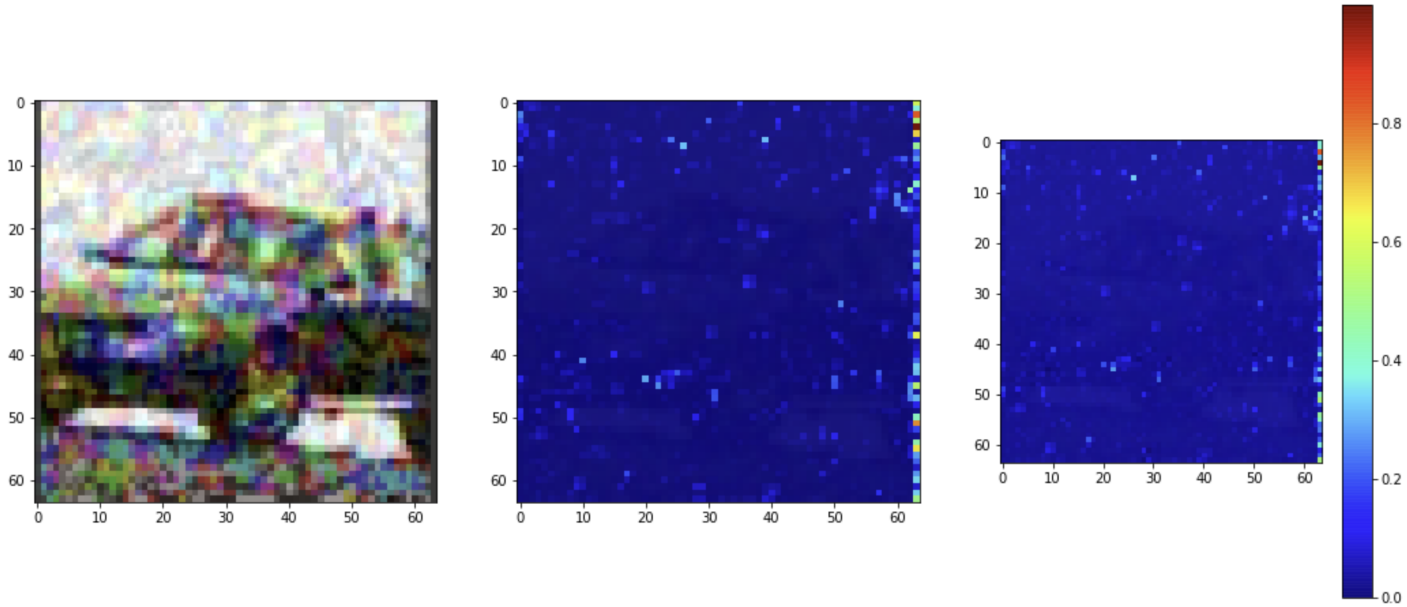
Adversarial Examples

We also visualized some of the adversarial training examples which we generated. Although some of the examples were too blurry for humans to recognize, the recognizable ones still show similar activations nullification.

We can see here that the adversarial examples were able to mask the activations indirectly by perturbing the inputs, resulting in misclassification. It is clear that the activations here do not resemble anything equal to the original object shape as seen earlier.



We see here that the activation and inhibitions becomes random and distorted. Unimportant parts becomes emphasized



We see here that the entire image's activation is obscured.

Additional Notes

Team Contributions

Arjun Patel [5%]:

- Research and considering off the shelf models for the base model.
- Outline and ideation for the project report.
- Ensuring the rubric elements covered in each part of the blog post.

- Clean up and clarity to repo readme for easy instructions for someone to get up and running.
- Saliency mapping collaboration and analysis in the report.
- Purpose, background, and goals of the project and research into the industry for breakthroughs and competition.
- Copy revision throughout the report to make it more blog-like.
- Solving initial bugs in processing for the first model attempted, environment-specific problems and issues with versioning of tools.

Kevin Lin [32.5%]:

- Led the general directions for the project (approaches, toolkits, models, training processes).
- Built the data pipeline for training and validation data.
- Incorporated data augmentation in data pipeline.
- Researched, designed, and tested models and hyperparameters.
- Training result visualization.
- Part 1-4 of the report.

Daniel Shin [57.5%]:

- Researched extensively on ways to combat data perturbation.
- Implemented the Adversarial Regularization and debugged in its entirety.
- Debugged numerous and extensive bugs in data preprocessing and original model. Including converting from categorical_entropy to sparse_categorical_entropy. And debugging validation data loader
- Deployed model on Google Cloud and other platforms
- Experimented with the NIPs model, https://github.com/luizgh/avc_nips_2018 (https://github.com/luizgh/avc_nips_2018)
- Posted and asked numerous questions on Piazza in search for resources and platforms.
- Attended office hours for theoretical questions and debugging issues
- Wrote the output pipeline for CSV as well as the input pipeline.
- Wrote 5. Combating Adversarial Conditions of this blog
- Changed model architecture from NASNET to mobilenetv2

Alan Chen [5%]:

- Wrote the code and resolved dependencies for older versions of keras-visualization
- Did majority of the work for Explainable AI

- Finished visualizations for saliency map outputs
- Finished the final csv pipeline, readme, and requirements
- Wrote the code for predicting specific samples and visualizing outputs
- Part 9 of the report and adversarial explainations.

References

Goodfellow, Ian, and Jonathon Shlens. “Explaining and Harnessing Adversarial Examples.” ArXiv.org (http://ArXiv.org), 20 Mar. 2015, arxiv.org/abs/1412.6572 (http://arxiv.org/abs/1412.6572).

Science, ODSC - Open Data. “Visualizing Your Convolutional Neural Network Predictions With Saliency Maps.” Medium, Medium, 21 June 2019, medium.com/@ODSC/visualizing-your-convolutional-neural-network-predictions-with-saliency-maps-9604eb03d766 (http://medium.com/@ODSC/visualizing-your-convolutional-neural-network-predictions-with-saliency-maps-9604eb03d766).