Computer Vision Assignment 1

22580530

April 2023

Question 1

I implement the CNN with PyTorch. I adapt code from a guide in the PyTorch documentation[2] to load the CIFAR-10 dataset and create a CNN with the same architecture as the one in the end of the lecture 1 slides. This network has the following hidden layers, from left to right:

- A convolutional layer with a 3×3 kernel and 32 channels.
- A convolutional layer with a 3×3 kernel and 64 channels.
- A fully connected layer with 64 neurons.

All layers use RelU activation, except for the output layer which uses softmax activation to give class probablities. To experiment with different variations of the network, I remove 20% of the training set to use as the validation set. I then examine how the following changes affect the training and validation error of the network:

1. Connecting a convolution layer with a 3×3 kernel and 128 channels to the last convolution layer.
2. Connecting a convolution layer with a 5×5 kernel and 16 channels before the first convolution layer.
3. Adding a 128 node fully connected layer before the output layer.
4. Using dropout with probability 0.2 on the output of each hidden layer.
5. Using dropout with probability 0.8 on the output of each hidden layer.
6. Using batch normalization on the output of each hidden layer.

The original model will be referred to as model 0, while the variants are named after their index in the above list. When training the models, I use a learning rate of 0.001. I use minibatch training with a batch size of 4 and train for two epochs (twice over the entire training set). The results are recorded in the following table:

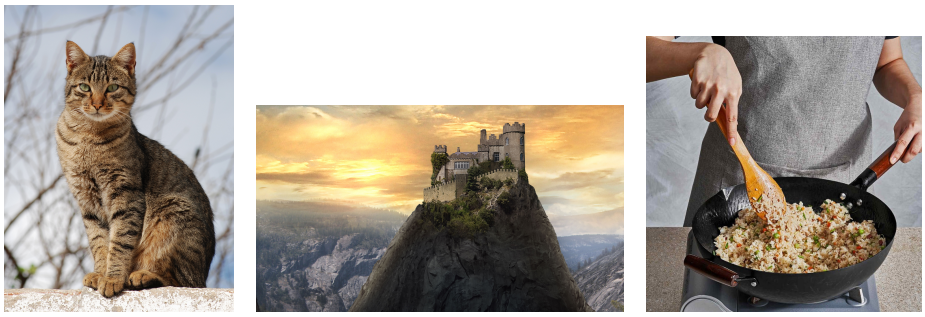
Model	Training accuracy	Validation accuracy
0	66%	63%
1	64%	62%
2	60%	57%
3	67%	62%
4	58%	57%
5	22%	22%
6	39%	40%

The original model achieved the highest validation accuracy. Models 1 and 2 show that adding convolutional layers did not significantly affect performance, implying that the original model is not limited by capacity. This is also suggested by the result of model 3, as adding an additional fully connected layer causes the model to overfit on the training set. From the results of models 4 and 5 it can be seen that a slight dropout slightly decreases accuracy, while a large dropout massively decreases accuracy. This implies that the original model is not overfitting the training set (and thus does not benefit from regularization). A similar conclusion can be made from model 6, which reduces accuracy from a different form of regularization (batch normalization).

Model 0, the original architecture, is evaluated on the test set and achieves an accuracy of 62%. This is similar to the validation accuracy and indicates that the model generalizes decently well.

Question 2

For this question I use AlexNet with pre-trained weights. Three images from the internet are used. They are shown in Figure 1, along with the probabilities of their classes predicted by AlexNet.



(a) Image A
"tabby" 46.7%

(b) Image B
"castle" 36.7%

(c) Image C
"wok" 83.4%

Figure 1: Test images for question 2

For a given image, I preprocess it into to a tensor of dimension $3 \times 224 \times 224$ compatible with AlexNet. An example is shown in Figure 2.

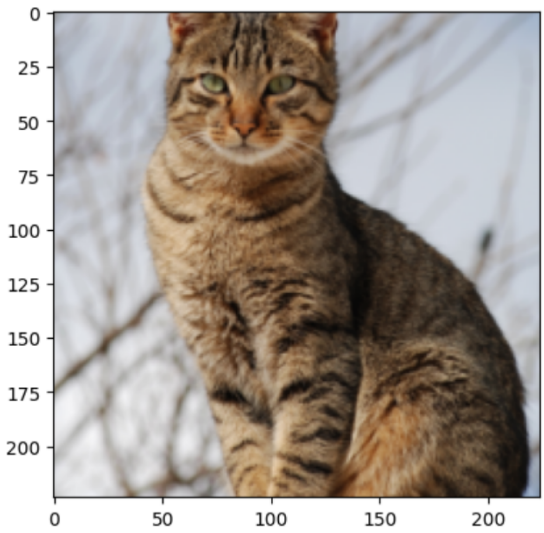


Figure 2: Image A after preprocessing

I define a function `occlude(x, y)` which occludes a portion of the image with a 60×60 grey square centred at a given position. This is done by making a copy of the image tensor and overwriting the values at relevant positions, as shown below:

```
new = data.clone().detach()
new[:,
    max(0, x-boxsize):min(data.size()[1], x+boxsize),
    max(0, y-boxsize):min(y+boxsize,data.size()[2])
    ] = 0.5
```

An example of applying the `occlude(x, y)` function is shown in Figure 3.

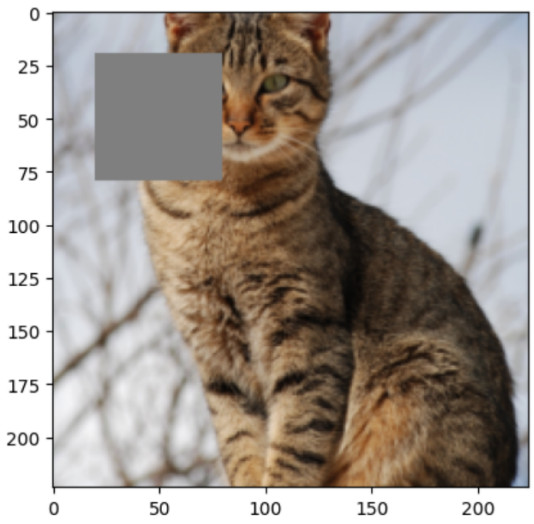
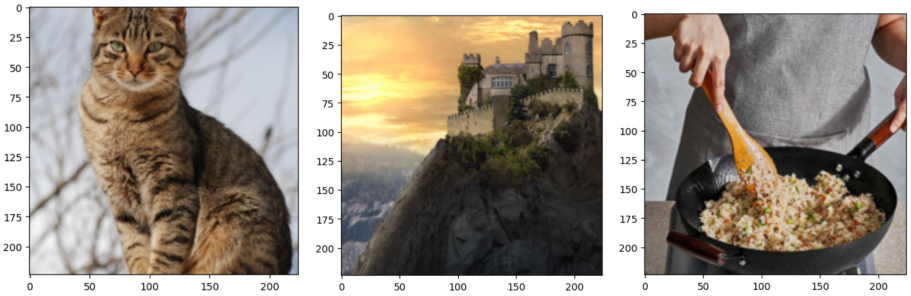


Figure 3: Image A after applying `occlude(50, 50)`

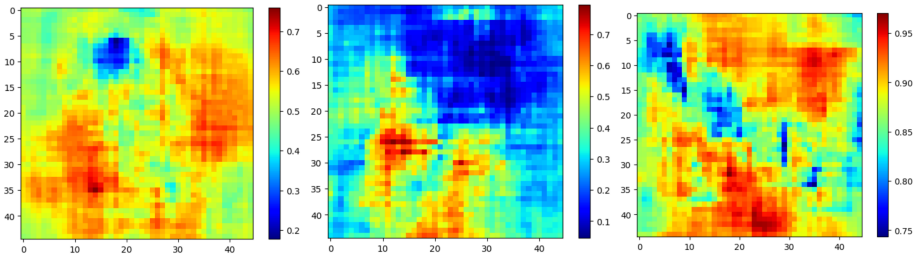
I slide this square over the image in steps of 5 and at each position, record AlexNet's output probability for the correct class. The measurements are collected into a saliency map.

Figure 4 shows the images after preprocessing, and Figure 5 shows the saliency map of each image.



(a) Image A (b) Image B (c) Image C

Figure 4: Images after preprocessing



(a) Image A (b) Image B (c) Image C

Figure 5: Saliency maps

The saliency map for image A shows that AlexNet relies heavily on the cat's face to classify it correctly, as there is a large drop in confidence when the face is occluded. The confidence significantly increases when certain portions of the background and the cat's body are covered. The tree branches, and the pattern on the cat's fur, may be different from that of the typical tabby cat in the model's training set.

As expected, occluding the castle in image B massively reduces the confidence in the "castle" class. The confidence also decreases when portions of the sky are occluded, indicating that this provides contextual information for this classification. Occluding portions of the mountain under the castle increases the confidence, likely because it reduces the probability of the conflicting "cliff" class.

Image C seems fairly robust to an occluded square of this size, as even the worst case position results in a confidently ($\sim 75\%$) correct classification. This may be because of the large size of the wok in the image, and the wealth of contextual clues (stirfry, cooking utensil). Surpirsingly, the greatest decreases in confidence occurs when the cook's hand or utensil are occluded. This indicates that contextual clues are important for classifying an image as a "wok".

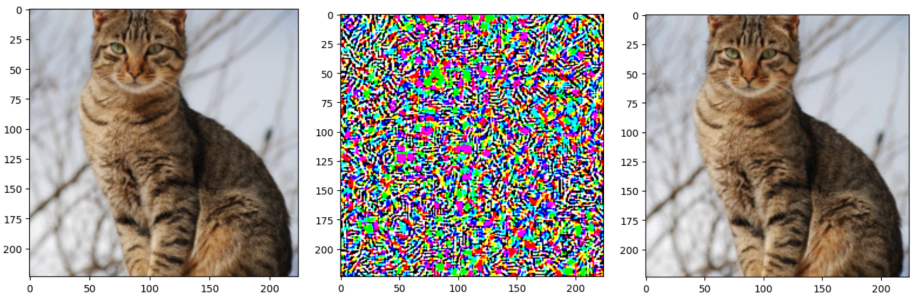
Question 3

I implement the fast gradient sign method in Pytorch and use it to generate adversarial images for AlexNet. I adapt code from a guide in the PyTorch documentation[1] to calculate the sign of the gradient.

In this question I use images A and B from question 2, as seen in Figure 1 (and similarly preprocessed as in Figure 4). For a given image, I run it through the neural network and calculate the loss. Then I use backpropagation to calculate the gradient with respect to the loss and compute the sign of this gradient. Finally, I calculate the adversarial example as a perturbation of the original image (stored in `data`):

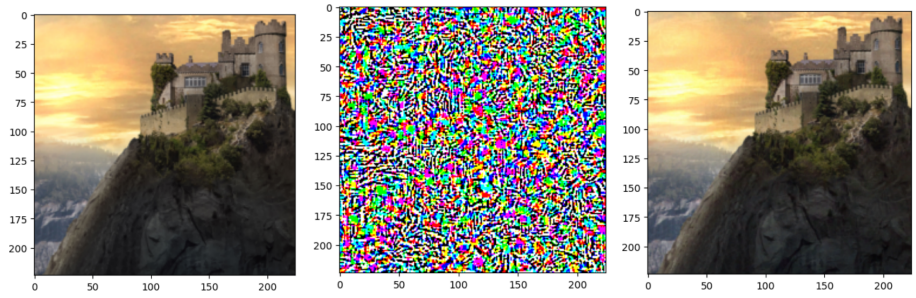
```
perturbed_data = data + epsilon*sign_data_grad
```

In Figure 6, each (preprocessed) image is shown with the sign of its gradient and the perturbated version, using `epsilon` $= 0.007$



(a) Original image (b) Sign of gradient (c) Perturbed image

Figure 6: Perturbation of image A



(a) Original image (b) Sign of gradient (c) Perturbed image

Figure 7: Perturbation of image B

Given the perturbed version of image A, the top three predicted classes are:

1. "ruffed grouse" with probability 13.0%
2. "tiger" with probability 7.7%
3. "tiger cat" with probability 6.3%

Similarly, the top classes for the perturbation of image B are:

1. "fur coat" with probability 6.1%
2. "totem pole" with probability 5.4%
3. "fountain" with probability 4.5%

The fast gradient sign method proves to be effective at confounding AlexNet with a change that is virtually imperceptible to the human eye. For both the adversarial examples, the correct class is not even in the top three most probable classifications by the model. However, the model does not have a high confidence for an incorrect class. This low incorrect class confidence may be due to the low value of `epsilon` used.

References

[1] *Adversarial Example Generation*. URL: `https://pytorch.org/tutorials/beginner/fgsm_tutorial.html`.

[2] *Training a Classifier*. URL: `https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html`.