Portland State UNIVERSITY

CS 4/510: Computer Vision & Deep Learning, Spring 2021

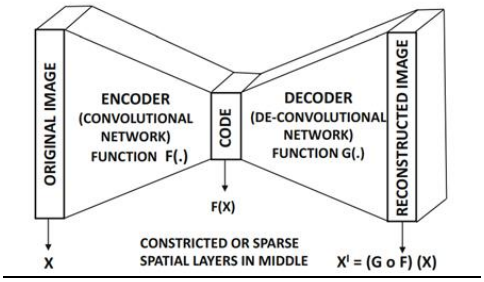
Programming Assignment #4

A. Rhodes

Note: This assignment is **due by Friday, 5/21 at 10:00pm**; turn in the assignment by email to our TA.

Note that there are (2) exercises listed below. **Complete one of the exercises of your choosing** (please don't do both).

Exercise #1: Convolutional Autoencoder



As described in lecture (see "Visualization" lecture series), in this exercise you will design and train a convolutional autoencoder (CAE) using the Fashion-MNIST dataset (example shown below): https://www.tensorflow.org/datasets/catalog/fashion_mnist; in Keras:

```
tf.keras.datasets.fashion_mnist.load_data()
```

Pre-process the data by dividing each image by 255.

Step 1: Design your CAE

For the basic CAE architecture, please use the basic template:

Encoder: Conv2D ⟶ Maxpooling ⟶ Conv2D ⟶ Maxpooling, etc.

Decoder: Conv2D ⟶ Upsampling2D ⟶ Conv2D ⟶ Upsampling, etc.

Please use a high-level DL library to construct your CAE, e.g., Keras/TF, etc. I recommend using RELU activation for all layers, with the exception of the last layer of the decoder sub-network, which should use a sigmoid activation. You are welcome to experiment with the width of each layer (i.e., number of conv filters), and the total number of layers (include at least three in each of the encoder/decoder sub-networks), but please architect your encoder and decoders **so that they are symmetric**. For example, if the first layer of your encoder has 32 filters, the last layer of your decoder should likewise have 32 filters. The encoder and decoder should have the same number of layers. **Include a model summary of your CAE in your assignment write-up**.

If you want to experiment with batch normalization, dropout, and other techniques we have discussed in class, feel free to do this – please include any relevant details about these design choices in your write-up.

Please read the following comments carefully:

***Note**: Design your CAE so that the last layer of your encoder is followed by a flatten operation. The middle layer of your model, i.e., the "**bottleneck layer**" (see the layer labeled "CODE" in the CAE diagram above) should be a **dense layer with two output neurons** (yes, only two) whose input is the flattened output of the encoder. In this way, the output of the encoder subnetwork is a 2D array.

The decoder portion of the CAE takes as input this 2D array; I recommend using a dense layer as the first layer of the encoder subnetwork (just as the previously described dense layer was the last layer of your encoder). Following the first dense layer in your decoder, you will need to **reshape the output** of this dense layer into a 3D tensor so that it can be passed successfully to a conv2D layer (corresponding with the last conv2D layer from your encoder).

For example, after the bottleneck layer we have a 2D vector. Suppose that the first layer of the encoder network is a Dense(2,4) layer (2 inputs, 4 outputs); you should then reshape this vector to, say, dimension (2,2,1) (still has four values), so that it can be passed to a conv2D layer, as the conv2D function will expect a 3D tensor.**

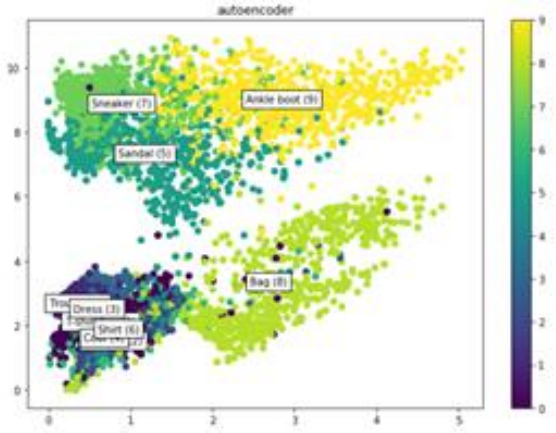
Part of the challenge of this exercise rests in designing the CAE architecture in a symmetric fashion (encoder/decoder have same number of layers and same width, i.e. number of corresponding filters); finally, you need to design the CAE so that the input and output are exactly the same dimension (otherwise we aren't reconstructing the input appropriately). To this end, for debugging purposes, I recommend checking/confirming the dimension of the

output of each layer of your CAE as you pass an image though it – feel free to use basic debug stop conditions as needed.

Step 2: When you have your CAE architected successfully, train your model using a standard optimizer (Adam, etc.), using **binary cross-entropy loss**; make sure that your loss is computed with respect to the input image and the reconstructed output of your CAE**. Include a graph of your training/test loss per epoch with your assignment write-up.** Please include several examples of test images and their corresponding reconstruction, generated by your model.

Step 3: Train a denoising CAE. Using your same CAE architecture from before, randomly add Gaussian noise, drawn from a standard normal distribution: $N(0,1)$ to your input and test images. Retrain your CAE using the noisy training data (but with the non-noisy images as the ground-truth for reconstruction). As in the previous step, use binary cross-entropy loss and **include a graph of your training/test loss per epoch**. Please include several examples of test images and their corresponding reconstruction, generated by your model.

Step 4: Pass each test image through your trained (non-denoising) CAE and extract the 2D latent feature from your encoder sub-network (if you don't want to run all of the test data, it is fine to use 500 or 1000 randomly sampled test datapoints). Plot all of these 2D points in graph, and color code them so that the color of each point corresponds with the ground-truth class label. Plot this 2D projection learned by your CAE to get a qualitative visualization of the dimensionality reduction process learned by your CAE. Your result should look something like the image below (results will vary depending on CAE architecture):



Include this 2D plot in your write-up; provide some insights in your write-up reflecting on this visualization.

Finally, compare the CAE-derived 2D projection of the Fashion-MNIST dataset to a standard dimensionality reduction algorithm: PCA. Please don't implement PCA from scratch; use a built-in library function.

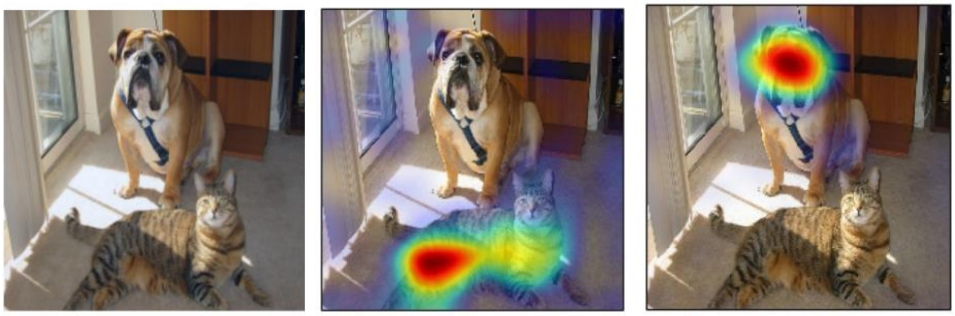
I personally recommend using the **scikit learn** library (https://scikit-learn.org). Write a for-loop over either the entire test dataset or a reasonably sized sample of 500 or 1000 test datapoints; for each test image **x**, simply execute:

```
from sklearn.decomposition import PCA

pca=PCA(n_components=2)
principalComponents = pca.fit_transform(x)
```

Which will yield the PCA-derived 2D compressed version of the test image **x**. As you did for the CAE-derived 2D projections, plot the PCA-derived 2D projection. Include this 2D plot in your write-up; provide some additional insights in your write-up reflecting on this visualization compared with the CAE-derived plot.

Exercise #2: Grad-CAM DL-Based Saliency



(a) Original Image (c) Grad-CAM 'Cat' (i) Grad-CAM 'Dog'

Dataset: Five test images are provided on D2L: gc1.jpg, …, gc5.jpg.

In this exercise you will recreate the core technique found in the well-known **Grad-CAM** saliency algorithm (full paper: https://arxiv.org/pdf/1610.02391.pdf) – note that you are not required to read the full paper to complete this exercise; below I summarize the essential steps.

Technique Summary:

Grad-CAM uses a pre-trained CNN to generate class-discriminative saliency maps for an input image. Concretely, the algorithm uses the gradient information from the final conv layer in the pre-trained CNN to generate these maps. Denote the kth activation map of the final convolutional layer: A^k .

In this exercise, I ask you to use a pre-trained VGG-16 architecture; the final conv layer activation maps from VGG-16 are of dimension 14×14, and there are a total of 512 such activation maps in this layer (one corresponding with each of the 512 filters in this layer).

To obtain the class discriminative localization map of dimension 14×14, we first compute the gradient of the score for class c (out of, say, the 1000 classes for Imagenet) which we

denote y^c – note that we **compute this gradient prior to the softmax operation** of the VGG network – with respect to the feature maps A^k.

These class-dependent gradients are computed for each of the 512 filters; then we average the gradient value with respect to each pixel in the 14×14 feature map. This average yields a scalar value (the "importance weight") of channel *c* with respect to feature map *k* denoted: α_k^c:

$$\alpha_k^c = \frac{1}{14 \cdot 14} \sum_{j=1}^{14} \sum_{i=1}^{14} \frac{\partial y^c}{\partial A_{ij}^k} \qquad (1)$$

After calculating α_k^c for the target class c, the authors compute a weighted combination of activation maps (using the $\alpha's$ as weights) and follow it with a RELU:

$$L_{Grad-CAM}^c = RELU\left(\sum_{k=1}^{512} \alpha_k^c A^K\right) \qquad (2)$$

This results in a coarse saliency map of dimension 14×14, which can then be resized to the original input image size to render a class-discriminative saliency map.

Step 1: Load a pre-trained (on Imagenet) VGG-16 model: https://keras.io/api/applications/vgg/ ; note that you should also import "**preprocess_input**" from the tensorflow.keras.applications.vgg16 library; pass each test image through **process_input**; this will ensure that your test images have undergone the exact same pre-processing technique that was used for ImageNet on your pre-trained VGG-16 model. Also import tensorflow.keras.applications.imagenet_utils.decode_predictions.

Step 2: Pass each test image through the pre-trained VGG-16 model – use model.predict(x); please print the top-3 predictions produced for each test image. To do so, you can execute:

```
prediction = model.predict(x)
print(decode_predictions(prediction,top=3))
```

Notice that "prediction" is a Numpy array of predictions. Please include the top-3 class predictions, e.g. "dog", "cat" produced by VGG-16 for each test image.

Step 3: Determine the argmax of the prediction vector from Step 2; this will give you the index of the top-1 predicted class generated by VGG for the input image. Extract the final conv layer of VGG-16, using (if using Keras) the "get_layer" function:

```
from keras import backend as K
last_layer=model.get_layer('block5_conv3')
gradient = K.gradients(model.output[:,ix],last_layer.output)[0]
```

Where above, "ix" is the index of the top-1 predicted class. If you are using a different DL software library, please refer to the documentation for this library on how to calculate this gradient.

Step 4: Above, gradient will give you a tensor of gradients of the top-1 channel output (per VGG16) with respect to the feature maps of the last convolutional layer of the network. From this tensor, calculate each α_k^c for each feature map, as shown in formula (1). Next, use this result and formula (2) to calculate $L^c_{\text{Grad-CAM}}$. Resize $L^c_{\text{Grad-CAM}}$ to the original input size of the image (use a built-in function to do this; preference: use bi-linear interpolation or comparable method).

Step 5: Include visualization of the input image, the top-1 predicted class, e.g., "bear", also include a visualization of the resized saliency heatmap $L^c_{\text{Grad-CAM}}$; finally, include a visualization of the saliency heatmap superimposed on top of the input image in your assignment write-up.

Repeat steps 3-5 for the test image, but this time, generate the saliency heatmap for the **second ranked class** of the top-3 generated by VGG-16; repeat this process for the **third ranked class** of the top-3. Repeat these trials for all the provided test images.

Report: Your report should include a short description of your experiments, along with the plots and discussion paragraphs requested above and any other relevant information to help shed light on your approach and results.

Here is what you need to turn in:

- Your report.
- Readable code.