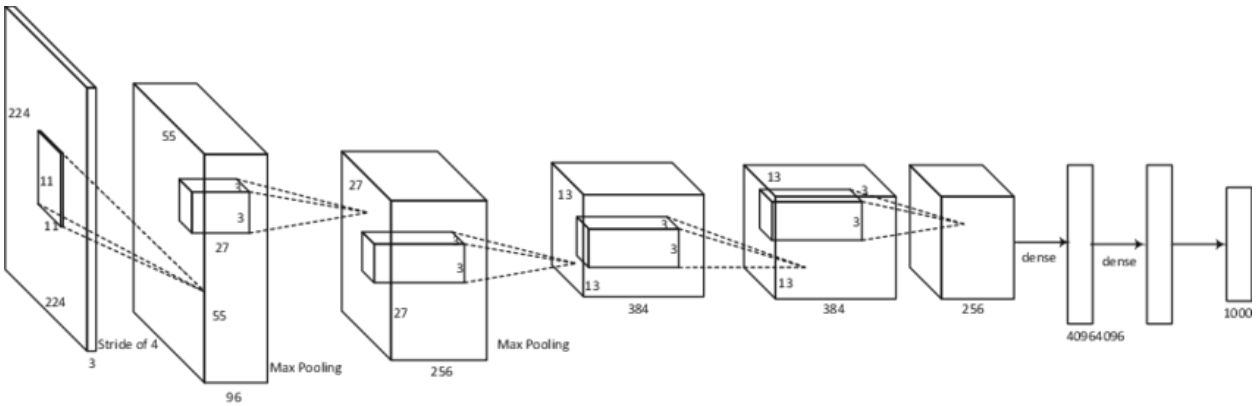
How can you decide on a network structure?

At this point, deciding on a network structure: how many layers to create, when to include dropout layers, and so on, may seem a bit like guessing, but there is a rationale behind defining a good model.

I think a lot of people (myself included) build up an intuition about how to structure a network from existing models. Take AlexNet as an example; linked is a nice, concise walkthrough of structure and reasoning.



AlexNet structure.

Preventing Overfitting

Often we see batch norm applied after early layers in the network, say after a set of conv/pool/activation steps since this normalization step is fairly quick and reduces the amount by which hidden weight values shift around. Dropout layers often come near the end of the network; placing them in between fully-connected layers for example can prevent any node in those layers from overly-dominating.

Convolutional and Pooling Layers

As far as conv/pool structure, I would again recommend looking at existing architectures, since many people have already done the work of throwing things together and seeing what works. In general, more layers = you can see more complex structures, but you should always consider the size and complexity of your training data (many layers may not be necessary for a simple task).

As You Learn

When you are first learning about CNN's for classification or any other task, you can improve your intuition about model design by approaching a simple task (such as clothing classification) and *quickly*trying out new approaches. You are encouraged to:

1. Change the number of convolutional layers and see what happens

2. Increase the size of convolutional kernels for larger images

3. Change loss/optimization functions to see how your model responds (especially change your **hyperparameters** such as learning rate and see what happens -- you will learn more about hyperparameters in the second module of this course)

4. Add layers to prevent overfitting

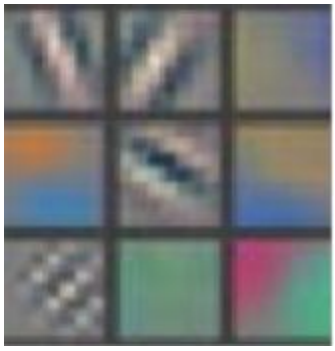
5. Change the batch_size of your data loader to see how larger batch sizes can affect your training

Always watch how **much** and how **quickly** your model loss decreases, and learn from improvements as well as mistakes!

Visualizing CNNs

Let's look at an example CNN to see how it works in action.

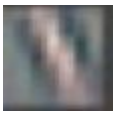
The CNN we will look at is trained on ImageNet as described in this paper by Zeiler and Fergus. In the images below (from the same paper), we'll see *what* each layer in this network detects and see *how*each layer detects more and more complex ideas.



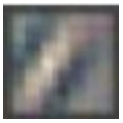
Example patterns that cause activations in the first layer of the network. These range from simple diagonal lines (top left) to green blobs (bottom middle).

The images above are from Matthew Zeiler and Rob Fergus' deep visualization toolbox, which lets us visualize what each layer in a CNN focuses on.

Each image in the above grid represents a pattern that causes the neurons in the first layer to activate - in other words, they are patterns that the first layer recognizes. The top left image shows a -45 degree line, while the middle top square shows a +45 degree line. These squares are shown below again for reference.

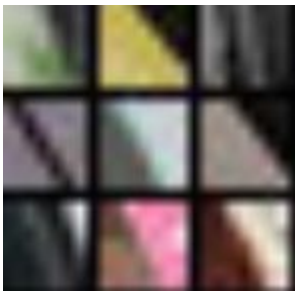


As visualized here, the first layer of the CNN can recognize -45 degree lines.



The first layer of the CNN is also able to recognize +45 degree lines, like the one above.

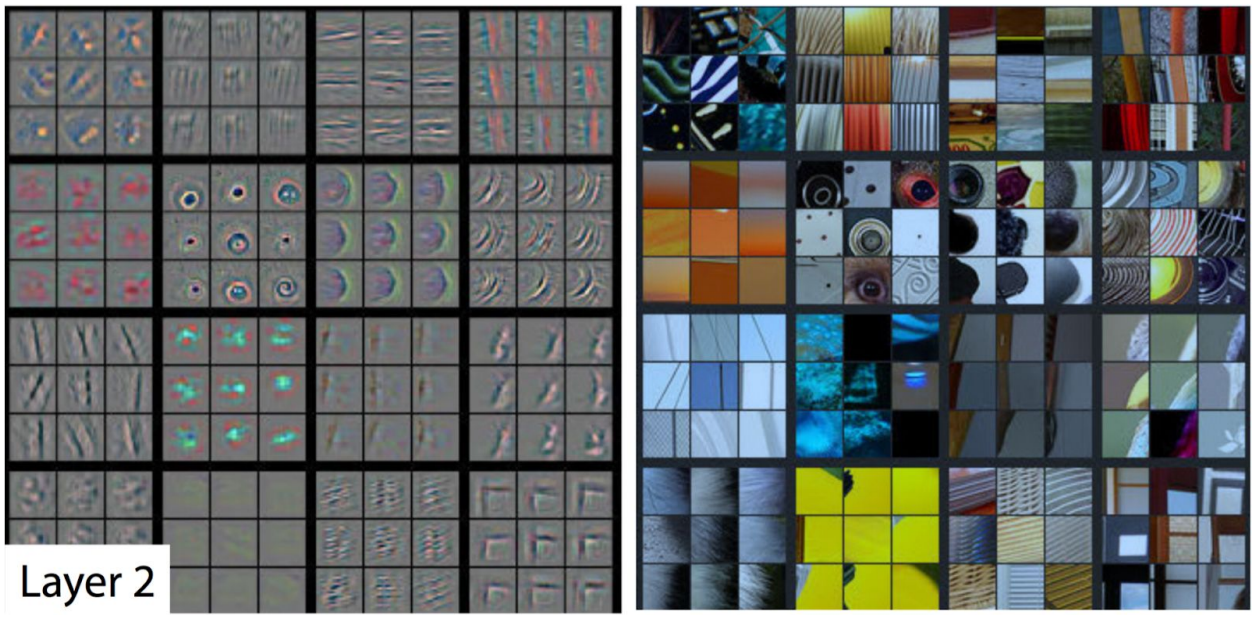
Let's now see some example images that cause such activations. The below grid of images all activated the -45 degree line. Notice how they are all selected despite the fact that they have different colors, gradients, and patterns.



Example patches that activate the -45 degree line detector in the first layer.

So, the first layer of our CNN clearly picks out very simple shapes and patterns like lines and blobs.

Layer 2



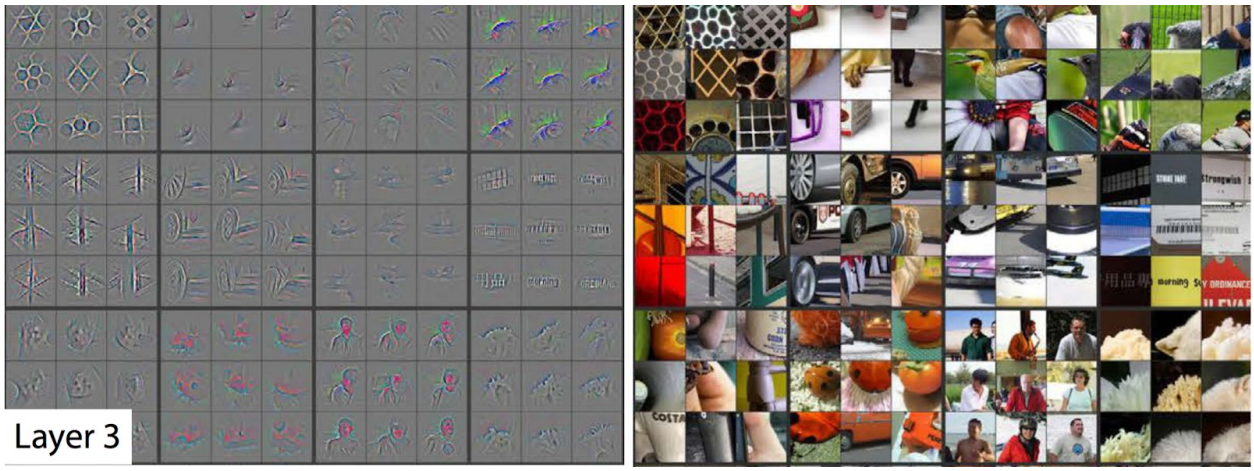
A visualization of the second layer in the CNN. Notice how we are picking up more complex ideas like circles and stripes. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

The second layer of the CNN captures complex ideas.

As you see in the image above, the second layer of the CNN recognizes circles (second row, second column), stripes (first row, second column), and rectangles (bottom right).

The CNN learns to do this on its own. There is no special instruction for the CNN to focus on more complex objects in deeper layers. That's just how it normally works out when you feed training data into a CNN.

Layer 3

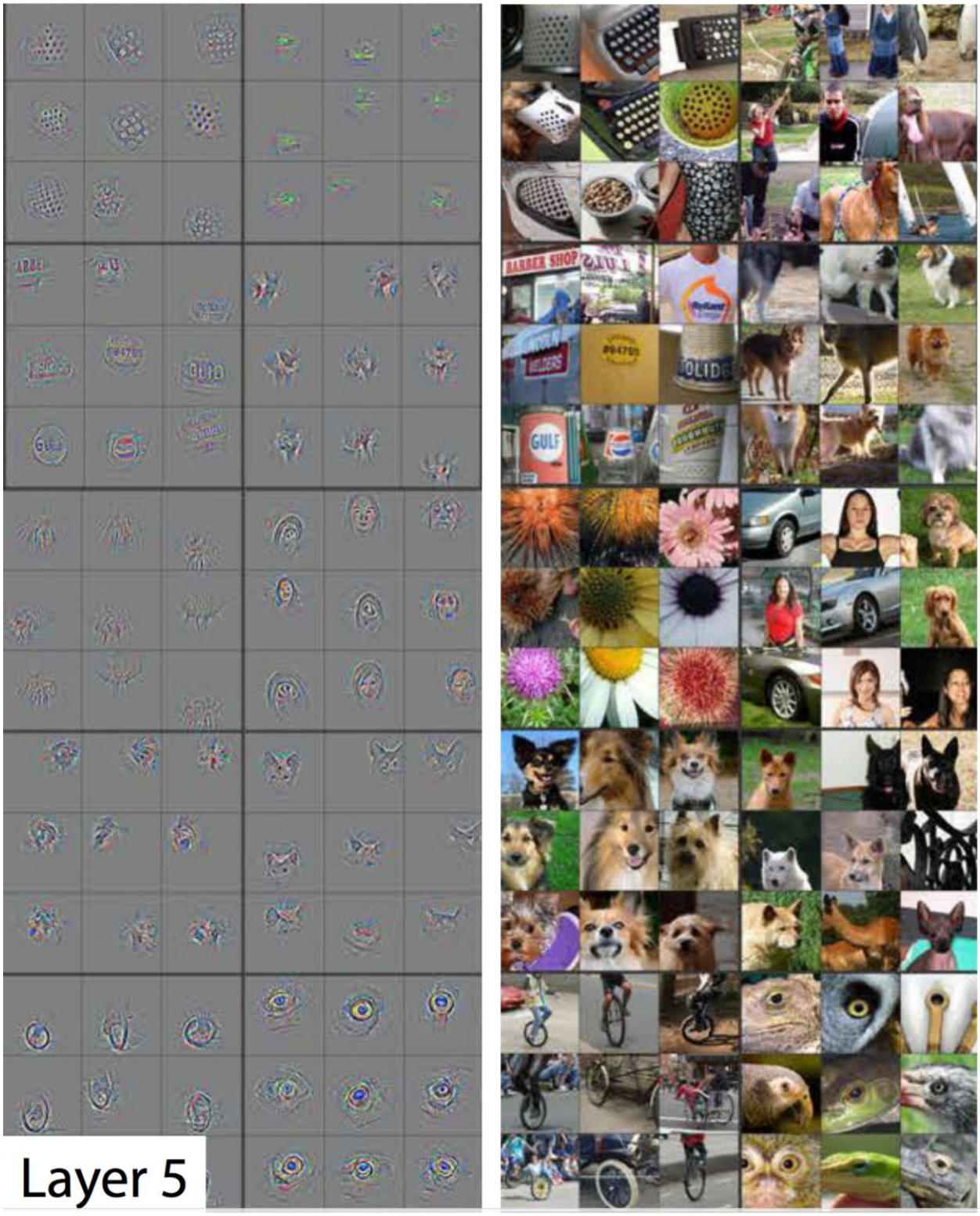


A visualization of the third layer in the CNN. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

The third layer picks out complex combinations of features from the second layer. These include things like grids, and honeycombs (top left), wheels (second row, second column), and even faces (third row, third column).

We'll skip layer 4, which continues this progression, and jump right to the fifth and final layer of this CNN.

Layer 5


Layer 5

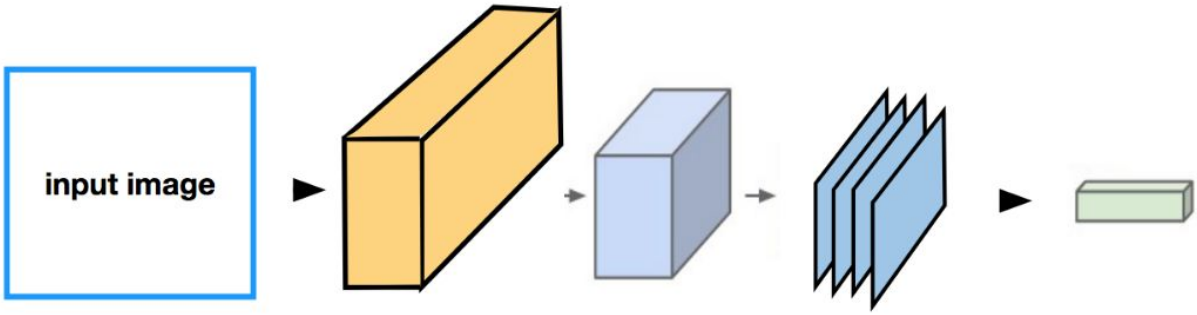
A visualization of the fifth and final layer of the CNN. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

The last layer picks out the highest order ideas that we care about for classification, like dog faces, bird faces, and bicycles.

Last Layer

In addition to looking at the first layer(s) of a CNN, we can take the opposite approach, and look at the last linear layer in a model.

We know that the output of a classification CNN, is a fully-connected class score layer, and one layer before that is a **feature vector that represents the content of the input image in some way**. This feature vector is produced after an input image has gone through all the layers in the CNN, and it contains enough distinguishing information to classify the image.



An input image going through some conv/pool layers and reaching a fully-connected layer. In between the feature maps and this fully-connected layer is a flattening step that creates a feature vector from the feature maps.

Final Feature Vector

So, how can we understand what's going on in this final feature vector? What kind of information has it distilled from an image?

To visualize what a vector represents about an image, we can compare it to other feature vectors, produced by the same CNN as it sees different input images. We can run a bunch of different images through a CNN and record the last feature vector for each image. This creates a feature space, where we can compare how similar these vectors are to one another.

We can measure vector-closeness by looking at the **nearest neighbors** in feature space. Nearest neighbors for an image is just an image that is near to it; that matches its pixels values as closely as possible. So, an image of an orange basketball will closely match other orange basketballs or even other orange, round shapes like an orange fruit, as seen below.

A basketball (left) and an orange (right) that are nearest neighbors in pixel space; these images have very similar colors and round shapes in the same x-y area.

Nearest neighbors in feature space

In feature space, the nearest neighbors for a given feature vector are the vectors that most closely match that one; we typically compare these with a metric like MSE or L1 distance. And *these* images may or may not have similar pixels, which the nearest-neighbor pixel images do; instead they have very similar content, which the feature vector has distilled.

In short, to visualize the last layer in a CNN, we ask: which feature vectors are closest to one another and which images do those correspond to?

And you can see an example of nearest neighbors in feature space, below; an image of a basketball that matches with other images of basketballs despite being a different color.

Nearest neighbors in feature space should represent the same kind of object.

Dimensionality reduction

Another method for visualizing this last layer in a CNN is to reduce the dimensionality of the final feature vector so that we can display it in 2D or 3D space.

For example, say we have a CNN that produces a 256-dimension vector (a list of 256 values). In this case, our task would be to reduce this 256-dimension vector into 2

dimensions that can then be plotted on an x-y axis. There are a few techniques that have been developed for compressing data like this.

Principal Component Analysis

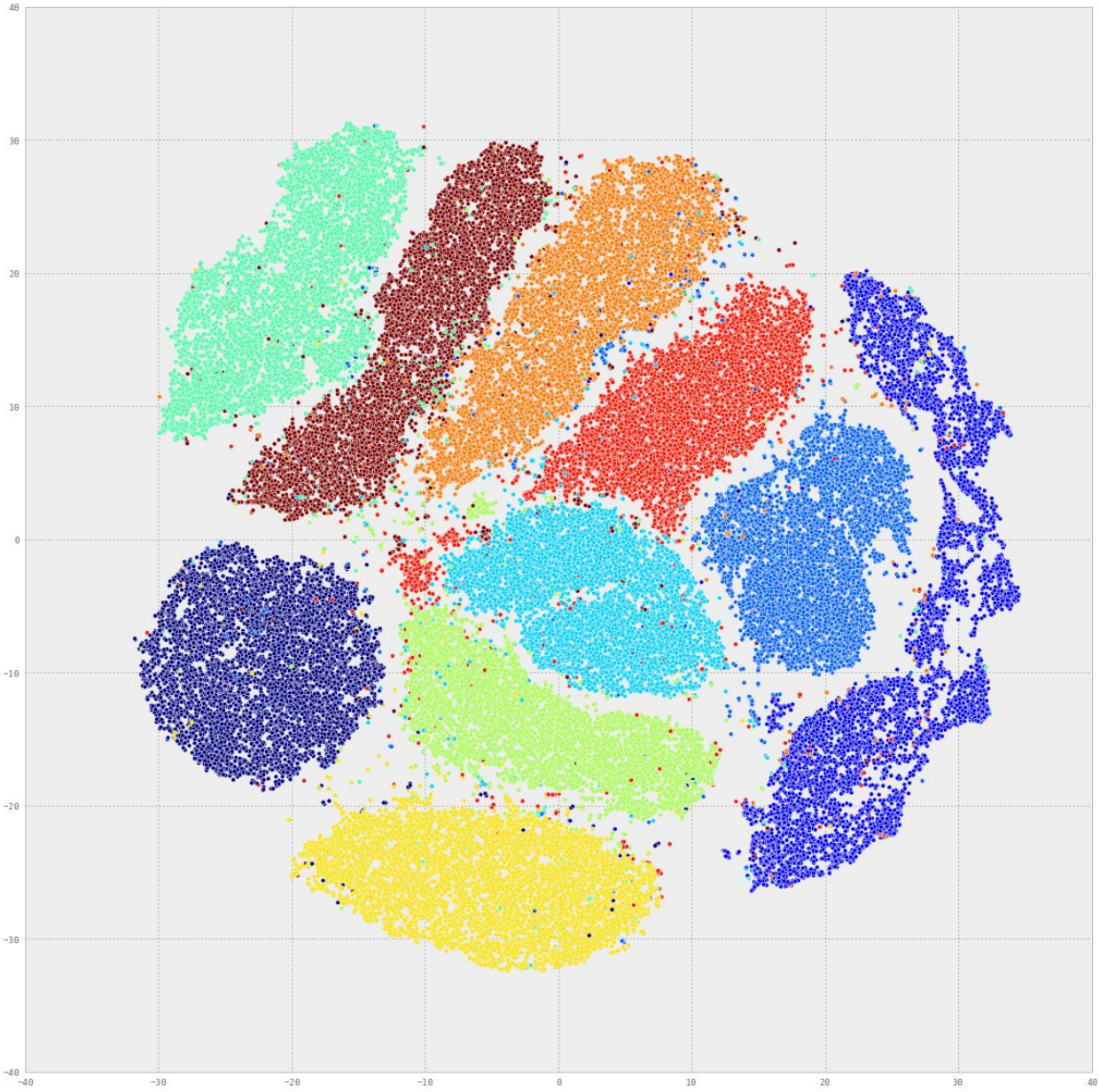
One is PCA, principal component analysis, which takes a high dimensional vector and compresses it down to two dimensions. It does this by looking at the feature space and creating two variables (x, y) that are functions of these features; these two variables want to be as different as possible, which means that the produced x and y end up separating the original feature data distribution by as large a margin as possible.

t-SNE

Another really powerful method for visualization is called t-SNE (pronounced, tea-SNEE), which stands for t-distributed stochastic neighbor embeddings. It's a non-linear dimensionality reduction that, again, aims to separate data in a way that clusters similar data close together and separates differing data.

As an example, below is a t-SNE reduction done on the MNIST dataset, which is a dataset of thousands of 28x28 images, similar to FashionMNIST, where each image is one of 10 hand-written digits 0-9.

The 28x28 pixel space of each digit is compressed to 2 dimensions by t-SNE and you can see that this produces ten clusters, one for each type of digits in the dataset!



t-SNE run on MNIST handwritten digit dataset. 10 clusters for 10 digits. You can see the generation code on Github.

t-SNE and practice with neural networks

If you are interested in learning more about neural networks, take a look at the **Elective Section: Text Sentiment Analysis**. Though this section is about text

classification and not images or visual data, the instructor, Andrew Trask, goes through the creation of a neural network step-by-step, including setting training parameters and changing his model when he sees unexpected loss results.

He also provides an example of t-SNE visualization for the sentiment of different words, so you can actually see whether certain words are typically negative or positive, which is really interesting!

This elective section will be especially good practice for the upcoming section Advanced Computer Vision and Deep Learning, which covers RNN's for analyzing sequences of data (like sequences of text). So, if you don't want to visit this section now, you're encouraged to look at it later on.

Other Feature Visualization Techniques

Feature visualization is an active area of research and before we move on, I'd like like to give you an overview of some of the techniques that you might see in research or try to implement on your own!

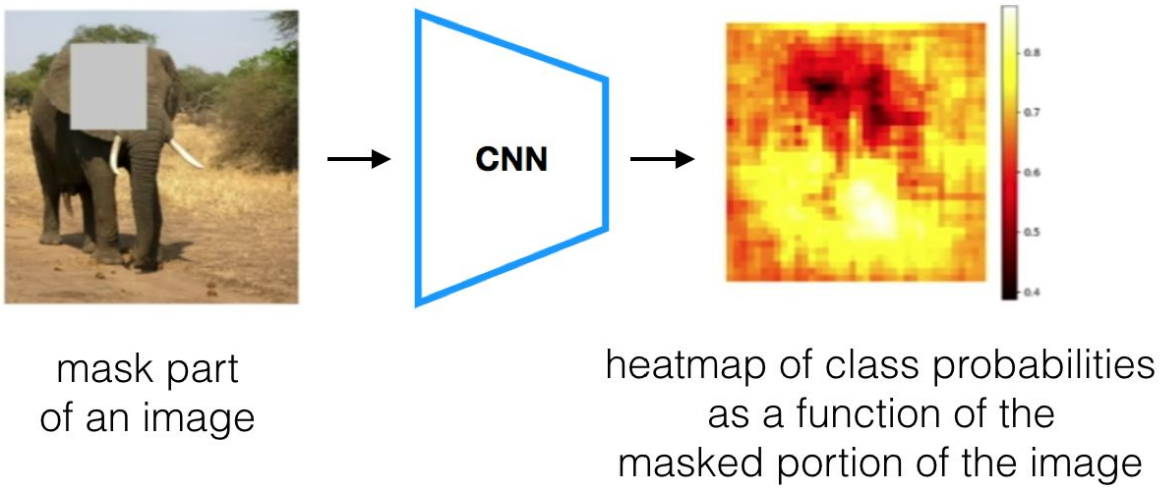
Occlusion Experiments

Occlusion means to block out or mask part of an image or object. For example, if you are looking at a person but their face is behind a book; this person's face is hidden (occluded). Occlusion can be used in feature visualization by blocking out selective parts of an image and seeing how a network responds.

The process for an occlusion experiment is as follows:

1. Mask part of an image before feeding it into a trained CNN,
2. Draw a heatmap of class scores for each masked image,
3. Slide the masked area to a different spot and repeat steps 1 and 2.

The result should be a heatmap that shows the predicted class of an image as a function of which part of an image was occluded. The reasoning is that **if the class score for a partially occluded image is different than the true class, then the occluded area was likely very important**!



Occlusion experiment with an image of an elephant.

Saliency Maps

Salience can be thought of as the importance of something, and for a given image, a saliency map asks: Which pixels are most important in classifying this image?

Not all pixels in an image are needed or relevant for classification. In the image of the elephant above, you don't need all the information in the image about the background and you may not even need all the detail about an elephant's skin texture; only the pixels that distinguish the elephant from any other animal are important.

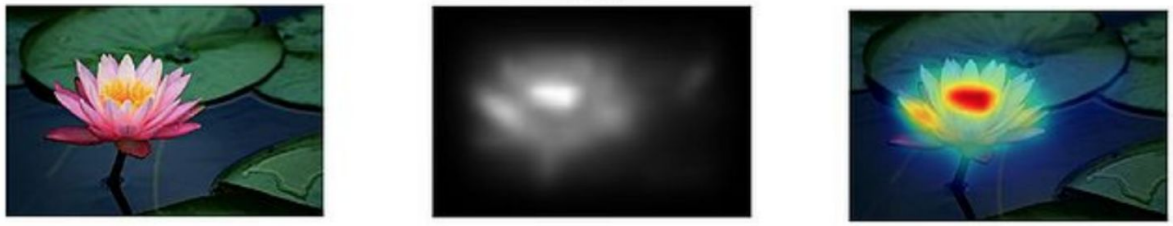
Saliency maps aim to show these important pictures by computing the gradient of the class score with respect to the image pixels. A gradient is a measure of change, and so,

the gradient of the class score with respect to the image pixels is a measure of how much a class score for an image changes if a pixel changes a little bit.

Measuring change

A saliency map tells us, for each pixel in an input image, if we change it's value slightly (by *dp*), how the class output will change. If the class scores change a lot, then the pixel that experienced a change, dp, is important in the classification task.

Looking at the saliency map below, you can see that it identifies the most important pixels in classifying an image of a flower. These kinds of maps have even been used to perform image segmentation (imagine the map overlay acting as an image mask)!

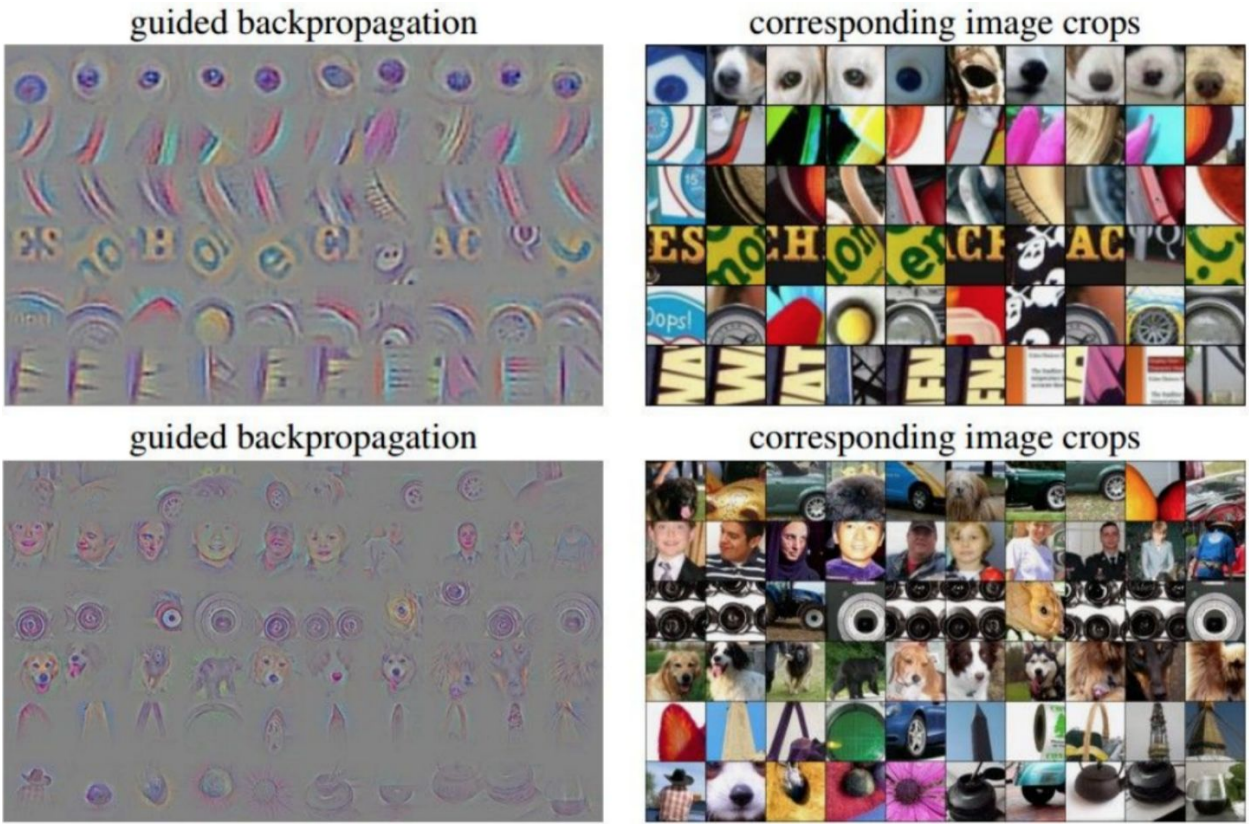


Graph-based saliency map for a flower; the most salient (important) pixels have been identified as the flower-center and petals.

Guided Backpropagation

Similar to the process for constructing a saliency map, you can compute the gradients for mid level neurons in a network with respect to the input pixels. Guided backpropagation looks at each pixel in an input image, and asks: if we change it's pixel value slightly, how will the output of a particular neuron or layer in the network change. If the expected output change a lot, then the pixel that experienced a change, is important to that particular layer.

This is very similar to the backpropagation steps for measuring the error between an input and output and propagating it back through a network. Guided backpropagation tells us exactly which parts of the image patches, that we've looked at, activate a specific neuron/layer.



Examples of guided backpropagation, from this paper.

Supporting Materials

Visualizing Conv Nets

Guided Backprop Network Simplicity

Important links:

- https://experiments.withgoogle.com/what-neural-nets-see
- https://www.youtube.com/watch?v=ghEmQSxT6tw

Deep Dream

DeepDream takes in an input image and uses the features in a trained CNN to amplifying the existing, detected features in the input image! The process is as follows:

1. Choose an input image, and choose a convolutional layer in the network whose features you want to amplify (the first layer will amplify simple edges and later layers will amplify more complex features).
2. Compute the activation maps for the input image at your chosen layer.
3. Set the gradient of the chosen layer equal to the activations and and use this to compute the gradient image.
4. Update the input image and repeat!

In step 3, by setting the gradient in the layer equal to the activation, we're telling that layer to give more weight to the features in the activation map. So, if a layer detects corners, then the corners in an input image will be amplified, and you can see such corners in the upper-right sky of the mountain image, below. For any layer, changing the gradient to be equal to the activations in that layer will amplify the features in the given image that the layer is responding to the most.

DeepDream on an image of a mountain.

Style Transfer

Style transfer aims to separate the content of an image from its style. So, how does it do this?

Isolating content

When Convolutional Neural Networks are trained to recognize objects, further layers in the network extract features that distill information about the content of an image and discard any extraneous information. That is, as we go deeper into a CNN, the input image is transformed into feature maps that increasingly care about the content of the image rather than any detail about the texture and color of pixels (which is something close to style).

You may hear features, in later layers of a network, referred to as a "content representation" of an image.

Isolating style

To isolate the style of an input image, a feature space designed to capture texture information is used. This space essentially looks at the correlations between feature maps in each layer of a network; the correlations give us an idea of texture and color information but leave out information about the arrangement of different objects in an image.

Combining style and content to create a new image

Style transfer takes in two images, and separates the content and style of each of those images. Then, to transfer the style of one image to another, it takes the content of the new image and applies the style of an another image (often a famous artwork).

The objects and shape arrangement of the new image is preserved, and the colors and textures (style) that make up the image are taken from another image. Below you can see an example of an image of a cat [content] being combined with the a Hokusai image of waves [style]. Effectively, style transfer renders the cat image in the style of the wave artwork.

If you'd like to try out Style Transfer on your own images, check out the **Elective Section: "Applications of Computer Vision and Deep Learning."**

Style transfer on an image of a cat and waves.