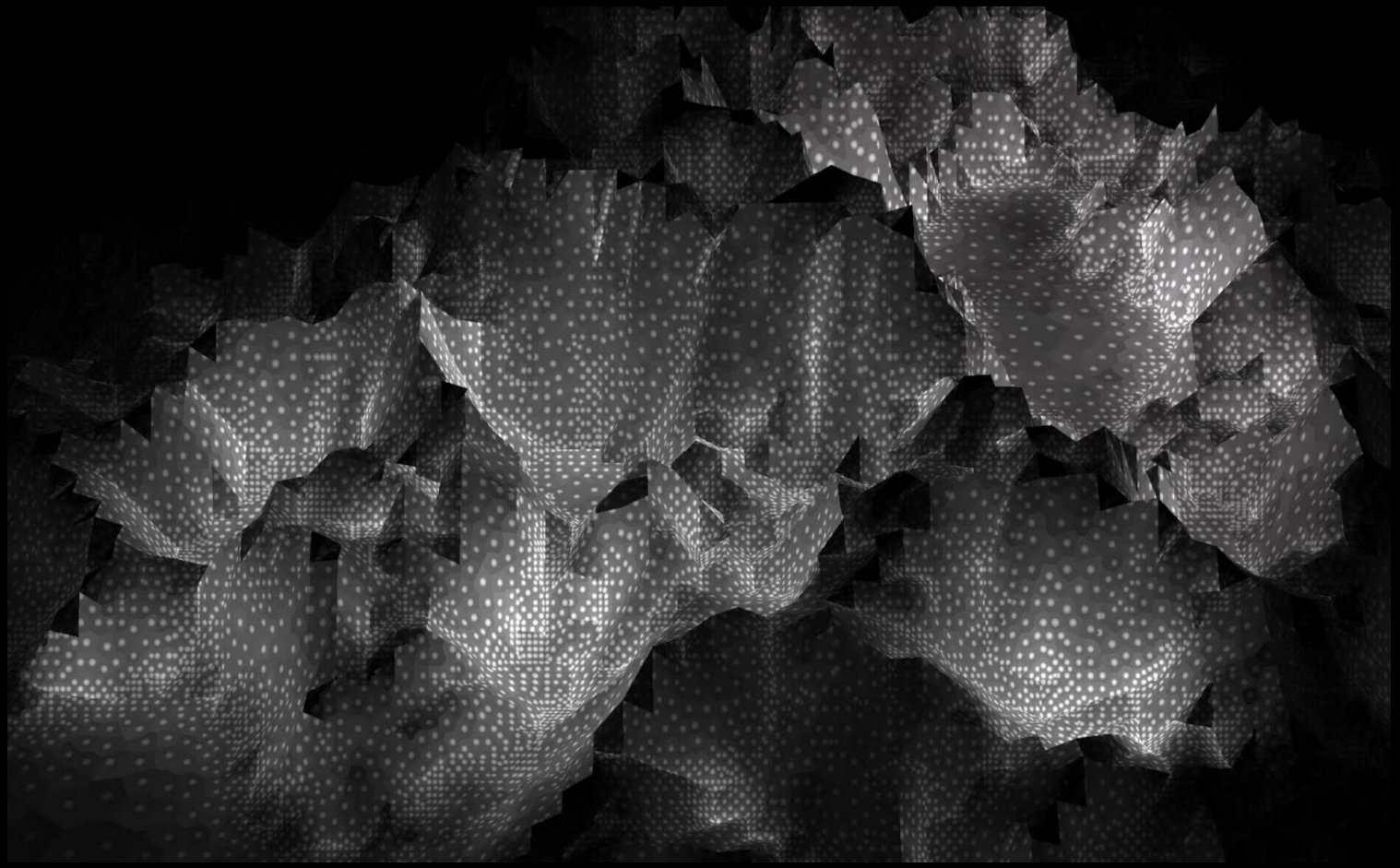


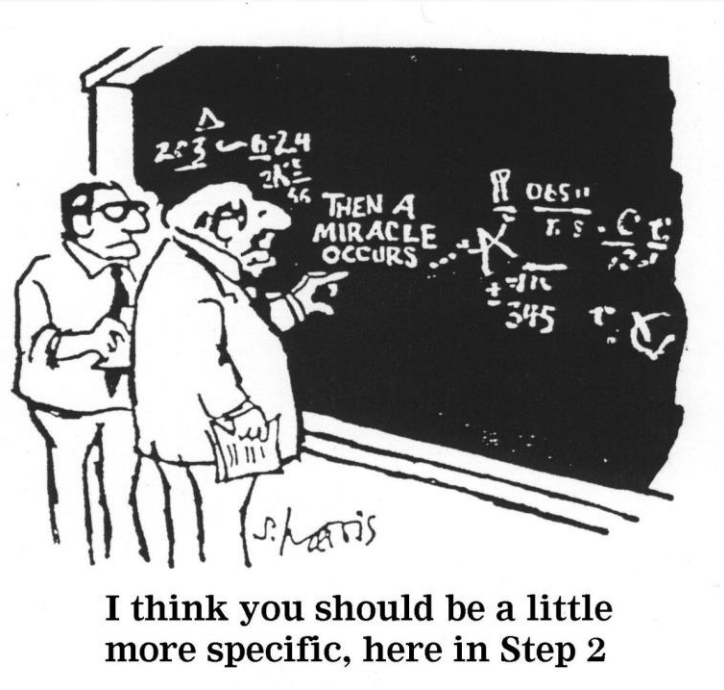
Visualization, Unsupervised Learning, and Attention
CS 410/510: CV & DL

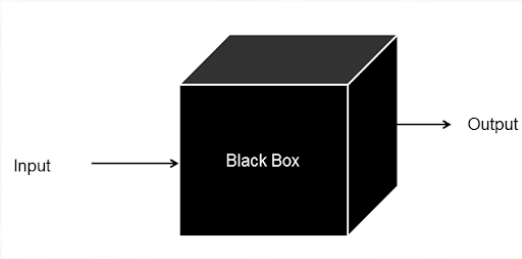
Outline

- Model Interpretability
- Autoencoders
- Saliency
- Feature Visualization; Image Inversion
- *DeepDream*
- Visual Attention: Captioning & Classification

Model Interpretability

• One of the common criticisms about NNs has been their **lack of interpretability**. Models that lack interpretability, i.e., **black box models**, often have limited use in real-world applications.



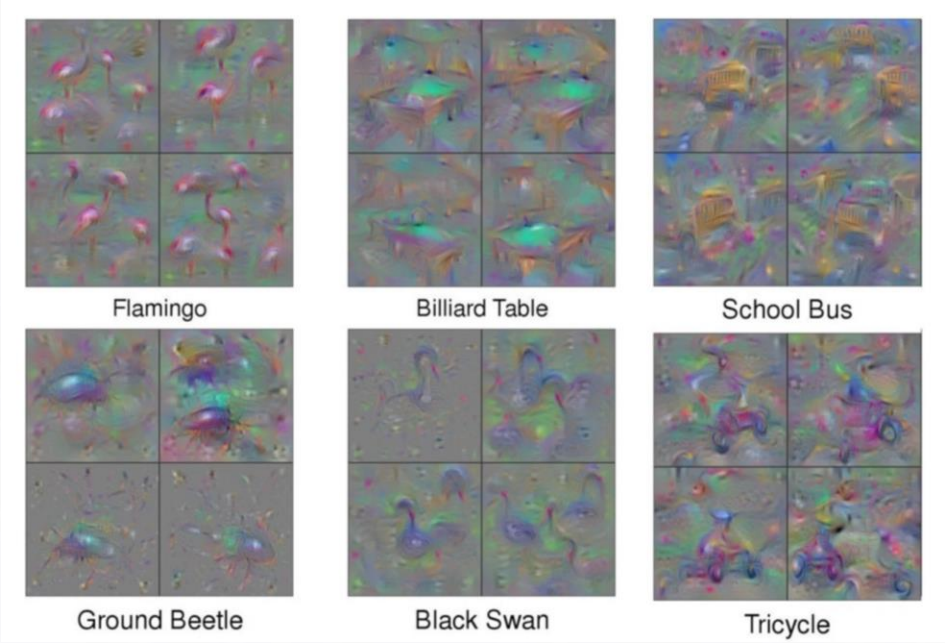


• DL systems lacking interpretability are inherently limited. Among other issues, **trust and verification** are difficult to achieve; the system itself can be **challenging to improve/learn** from; it may be difficult to detect and ameliorate the **presence of model bias**; there may even be unforeseen **legal consequences** to using such as system.

Model Interpretability

• However, as it turns out, one can use backpropagation to determine the features which contribute most significantly to the classification of a particular test instance.

• This method provides the analyst with an understanding of the importance of different features learned by the model; moreover, this approach can also be used as a mechanism for **feature selection**.



Model Interpretability

• For instance, consider a test datum $\boldsymbol{x} = \langle x_1, \ldots, x_d \rangle$ (an image, for example), for which the multilabel output scores of the NN are $o_1, \ldots, o_k$. Furthermore, let the output of the winning class among the k outputs be $o_m, 1 \leq m \leq k$.

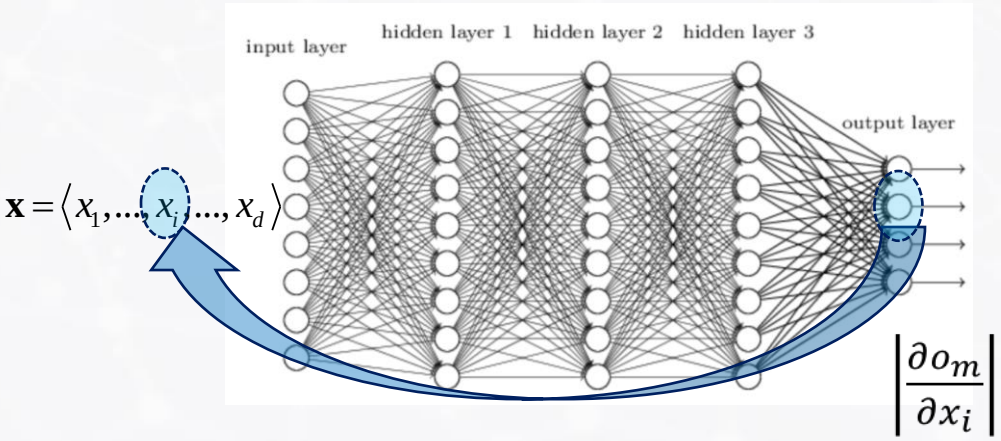
• Our goal is to identify the **features that are most pertinent to the classification** of this test instance. Moreover, we wish to identify the specific sensitivity of the output o_m to each x_i.

Model Interpretability

• For instance, consider a test datum $\boldsymbol{x} = \langle x_1, \ldots, x_d \rangle$ (an image, for example), for which the multilabel output scores of the NN are $o_1, \ldots, o_k$. Furthermore, let the output of the winning class among the k outputs be $o_m, 1 \leq m \leq k$.

• Our goal is to identify the **features that are most pertinent to the classification** of this test instance. Moreover, we wish to identify the specific sensitivity of the output o_m to each x_i.

• Features with **large absolute magnitudes** of this sensitivity are naturally relevant to the classification of this test instance and will have the greatest influence on the classification for the winning class. Thus, we need to compute: $\left|\frac{\partial o_m}{\partial x_i}\right|$.



Model Interpretability

- Thus, we need to compute: $\left|\frac{\partial o_m}{\partial x_i}\right|$.

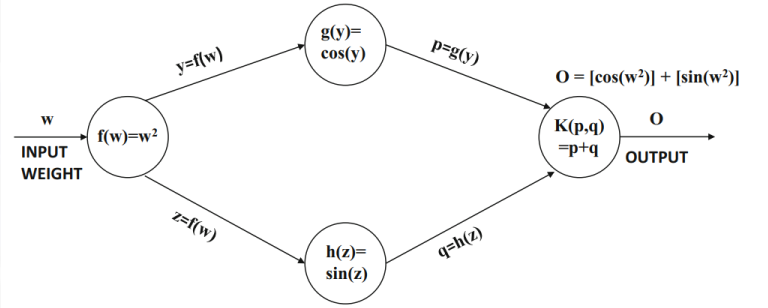
- Notice that the **sign of the derivative** also reveals whether increasing x_i slightly from its current value increases or decreases the score of the winning class. Other partials derivatives for non winning classes can also be relevant, but their importance is diminished when many classes exist.

Model Interpretability

• Thus, we need to compute: $\left|\frac{\partial o_m}{\partial x_i}\right|$.

• Notice that the **sign of the derivative** also reveals whether increasing x_i slightly from its current value increases or decreases the score of the winning class. Other partials derivatives for non winning classes can also be relevant, but their importance is diminished when many classes exist.

• How do we compute this derivative? Fortunately, it is a straightforward application of backpropagation. Features with largest aggregate sensitivity over entire training data are most relevant.



$$\frac{\partial o}{\partial w} = \underbrace{\frac{\partial K(p,q)}{\partial p}}_{1} \cdot \underbrace{g'(y)}_{-\sin(y)} \cdot \underbrace{f'(w)}_{2w} + \underbrace{\frac{\partial K(p,q)}{\partial q}}_{1} \cdot \underbrace{h'(z)}_{\cos(z)} \cdot \underbrace{f'(w)}_{2w}$$
$$= -2w \cdot \sin(y) + 2w \cdot \cos(z)$$
$$= -2w \cdot \sin(w^2) + 2w \cdot \cos(w^2)$$

• This process is closely related to "**saliency analysis**" in classical CV, which tells us which features (pixels) in an image are most relevant in the image being classified as a dog, etc.

Autoencoders

•**Autoencoders** (AEs) represent a fundamental architecture that is used for various kinds of unsupervised learning, including feature engineering, principal component analysis (PCA), incomplete data factorization, and dimensionality reduction.

• One significant advantage of AEs that is often overlooked, includes their strong conceptual consonance with matrix factorization methods. From this perspective, one can frame a variety of traditional matrix factorization methods (e.g., sparse factorization, PCA, etc.), as a special case – through the use of bespoke architectures – of AEs.

Autoencoders

•**Autoencoders** (AEs) represent a fundamental architecture that is used for various kinds of unsupervised learning, including feature engineering, principal component analysis (PCA), incomplete data factorization, and dimensionality reduction.

• One significant advantage of AEs that is often overlooked, includes their strong conceptual consonance with matrix factorization methods. From this perspective, one can frame a variety of traditional matrix factorization methods (e.g., sparse factorization, PCA, etc.), as a special case – through the use of bespoke architectures – of AEs.

For this brief digression about AEs, we show:

(1) Classical dimensionality reduction methods (SVD) are special cases of neural architectures (!)

(2) By modulating the basic AE architecture, one can generate complex non-linear embeddings of data. In particular, neural architectures provide unprecedented flexibility in controlling properties of these embeddings.

Autoencoders

• Autoencoders aim to **reconstruct input data** by passing it through a network bottleneck. This network bottleneck serves as a mechanism for (automated) dimensionality reduction.

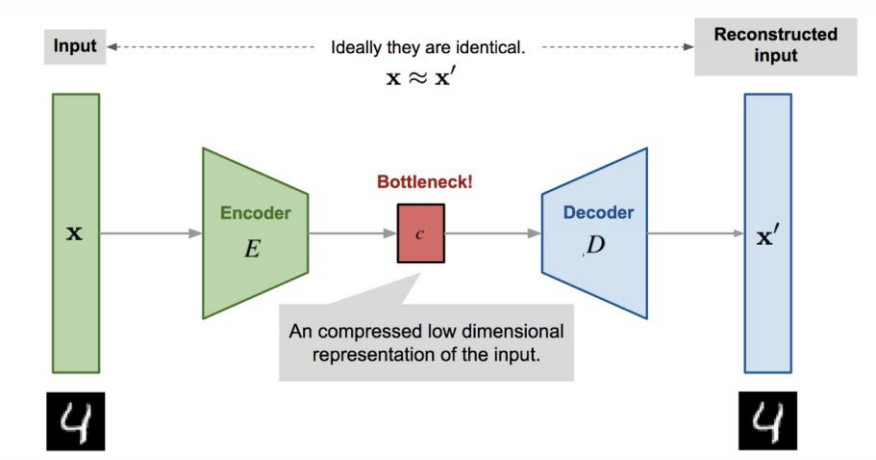
• Although reconstructing data might seem like a trivial matter in general, the introduction of a bottleneck holds a reduced representation of the data, and hence the final layer of the AE cannot, in general, reconstruct the input data exactly. Thus, this type of reconstruction is inherently **lossy**.

Autoencoders

• Autoencoders aim to **reconstruct input data** by passing it through a network bottleneck. This network bottleneck serves as a mechanism for (automated) dimensionality reduction.

• Although reconstructing data might seem like a trivial matter in general, the introduction of a bottleneck holds a reduced representation of the data, and hence the final layer of the AE cannot, in general, reconstruct the input data exactly. Thus, this type of reconstruction is inherently **lossy**.

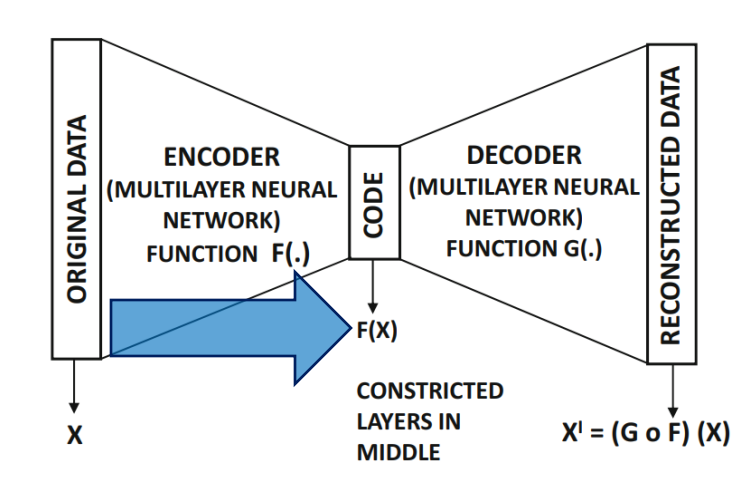
• The **loss function** of the AE is typically a sum-of-squares (note that **cross-entropy** and **binary cross-entropy** are also commonly used) difference between the input and output, in order to enforce veracity of the reconstruction. Note that the innermost hidden layer will be hierarchically related to those in the other hidden layers.



Autoencoders

• AEs commonly have a **symmetrical structure**, as shown. The subnetwork preceding the bottleneck layer represents the **Encoder network**, and the subnetwork following the bottleneck represents the **Decoder network** (we will see this schematic again in our course).

• If we denote the network input $\boldsymbol{x} \in D$ (where D is the data distribution) and the encoder $F(\cdot)$, respectively, then $F(\boldsymbol{x})$, symbolizes a reduced representation of $\boldsymbol{x}$ per the network latent space.

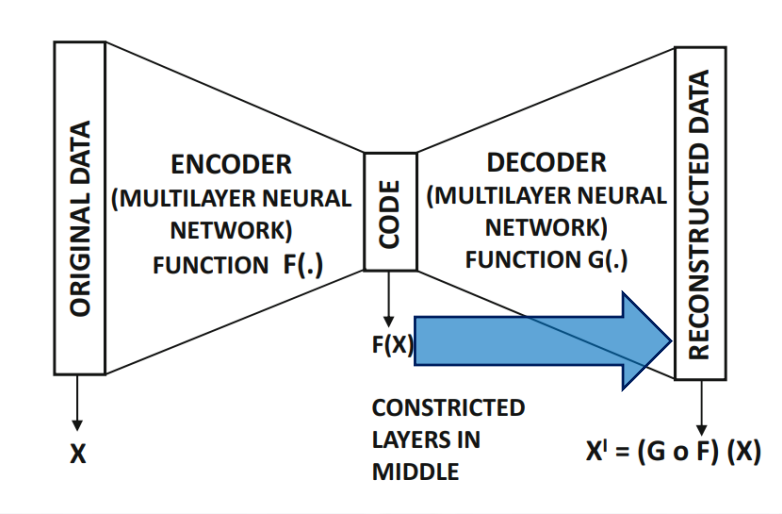


Autoencoders

• Conversely, if we sample a point from the network latent space, i.e. $F(\acute{x})$ and subsequently pass this point through the decoder subnetwork $G(\cdot)$, we generate a **synthetic datum** – this is to say, a datum resembling a sample from the distribution D.

• Together, the subnetworks of an AE form a potent tandem. The encoder subnetwork functions as a hierarchical dimensionality reduction mechanism, while the decoder subnetwork provides an apparatus to generate synthetic data.

• An additional benefit of AEs is that they can be trained in an **unsupervised** fashion! With reconstruction loss no ostensible "label" is required.



Autoencoders: SVD

•Next, we describe an AE with a single hidden layer to better illustrate the conceptual link between AEs and SVD.

• We devise an AE with d input dimensions and a single hidden layer of $\mathrm{k} \ll \mathrm{d}$. Assume we have an $n \times d$ data matrix D and we wish to factorize it into the product of an $n \times k$ matrix U and a $\mathrm{d} \times k$ matrix V:

$$D \approx UV^T$$

• Here k is the rank of the factorization; U contains he reduced representation of the data, and the matrix V contains the basis vectors.

Autoencoders: SVD

•Next, we describe an AE with a single hidden layer to better illustrate the conceptual link between AEs and SVD.

• We devise an AE with d input dimensions and a single hidden layer of $\mathrm{k} \ll \mathrm{d}$. Assume we have an $n \times d$ data matrix D and we wish to factorize it into the product of an $n \times k$ matrix U and a $\mathrm{d} \times k$ matrix V:

$$D \approx UV^T$$

• Here k is the rank of the factorization; U contains he reduced representation of the data, and the matrix V contains the basis vectors.

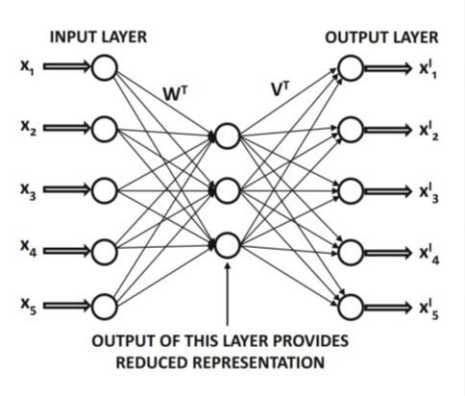
• In traditional ML, matrix factorization is solved by minimizing the **Frobenius norm** (element-wise L2 norm) of the residual matrix: $D - UV^T$. This yields the following objective:

$$\arg\min_{U,V} \left\| D - UV^T \right\|_F^2$$

Autoencoders: SVD

$$\arg\min_{U,V} \left\| D - UV^T \right\|_F^2$$

• Notice that the above optimization problem is relatively easy to solve using gradient-descent based algorithms. However, in this instance, we demonstrate how to encapsulate this matrix factorization problem using a neural network framework.



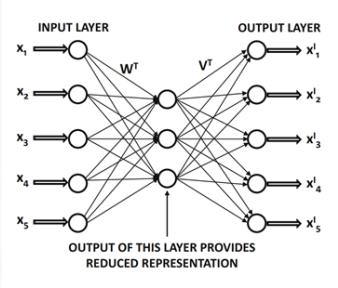
• The neural architecture shown has k hidden units; the encoder weights are contained in the $\mathrm{d} \times k$ matrix denoted by W.

• Notice that the AE depicted creates the reconstructed representation $\mathrm{D}W^T V^T$ of the original data matrix, where we use the reconstruction loss:

$$\left\| DW^T V^T - D \right\|^2$$

Autoencoders: SVD

$$\left\| DW^T V^T - D \right\|^2$$

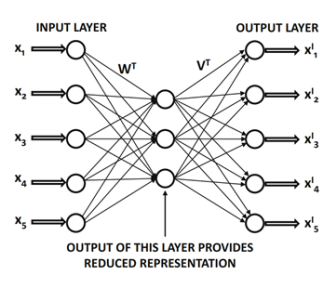


• The optimal solution to this problem is given by: $W = \left(V^T V\right)^{-1} V^T$ (this formulation is known as the **pseudo-inverse** of V):

$$\left\| DW^T V^T - D \right\|^2 = \left\| D\left(\left(V^T V\right)^{-1} V^T\right)^T V^T - D \right\|^2 = \left\| DV\left(V^T V\right)^{-1} V^T - D \right\|^2 = \left\| DVV^{-1}\left(V^T\right)^{-1} V^T - D \right\|^2 = 0$$

Autoencoders: SVD

$$\left\| DW^T V^T - D \right\|^2$$



• The optimal solution to this problem is given by: $W = \left(V^T V\right)^{-1} V^T$ (this formulation is known as the **pseudo-inverse** of V):

$$\left\| DW^T V^T - D \right\|^2 = \left\| D\left(\left(V^T V\right)^{-1} V^T\right)^T V^T - D \right\|^2 = \left\| DV\left(V^T V\right)^{-1} V^T - D \right\|^2 = \left\| DVV^{-1}\left(V^T\right)^{-1} V^T - D \right\|^2 = 0$$

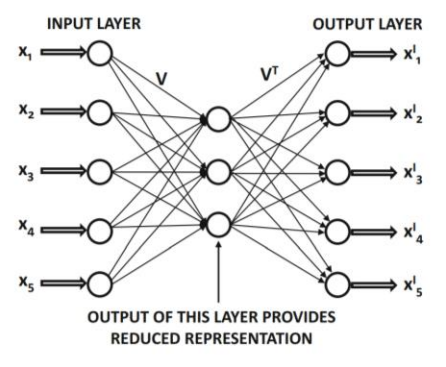
• Notice that the AE learning process **might deviate from this condition** if the D matrix has a small rank, etc. Post-multiplying the input D by W^T yields: $DW^T \approx UV^T V\left(V^T V\right)^{-1} = U$; this shows that we get the reduced representation from the matrix factorization.

• SVD yields factorization UV^T where the columns of V are orthonormal. The loss function of the AE allows for discovering alternative minima, so it is possible to find an optimal solution in which the columns of V are not necessarily mutually orthogonal or unit length. Nevertheless, the subspace spanned by the k columns of V will be the same as that spanned by the top-k basis vectors of SVD.

Autoencoders: SVD

• Many variants of AEs exist. For instance, one can employ **weight-sharing**, where some of the weights between the encoder and decoder are shared. This approach has the effect of reducing the parameter footprint of the AE and simultaneously enforcing regularization, and thus reducing overfitting.

• In the most straightforward instance of weight-sharing for AEs, in a one hidden layer variant $W = V^T$ (as shown).



• Alternatively, one can add a **sparse penalty** (i.e. L1-penalty) for approximation of a sparse matrix factorization:

$$\arg\min_{W,V} \left\| DW^T V^T - D \right\|_F^2 + \left\| DW^T \right\|_1$$

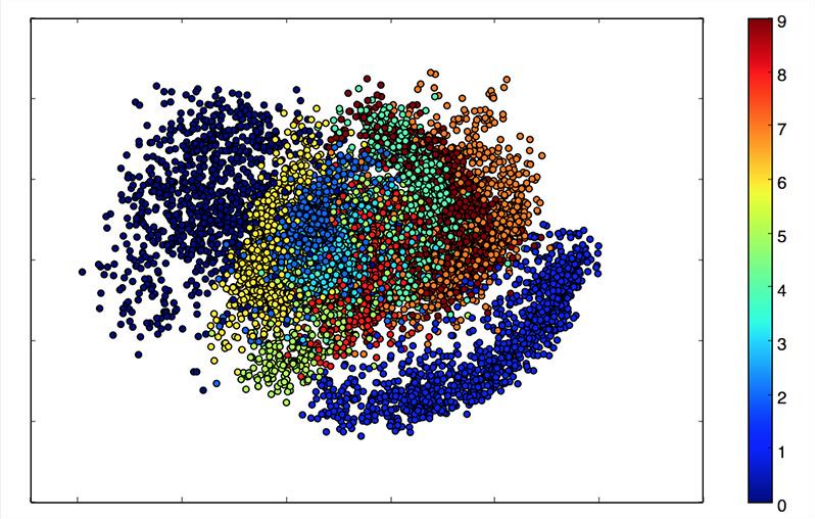
• A significant benefit of formulating matrix factorization via neural networks is that we only need to augment the loss function (here adding the L1 regularization term) in order to solve the sparse variant of matrix factorization.

Autoencoders: SVD

• The real power of AEs is realized when deeper variants are used (along with non-linear activations).

• As we know, DNNs provide an extraordinary amount of representational power. Multiple layers provide hierarchically reduced representations of data.

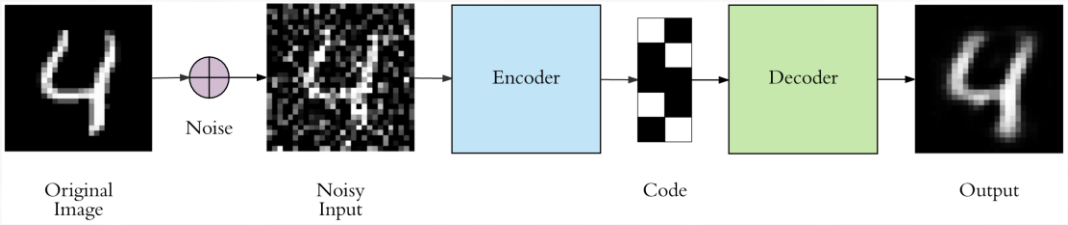
• Using deep AEs, It is possible to achieve extremely compact data reductions. Greater reduction is always achieved by using non-linear units, which implicitly map warped manifolds into hyperplanes; such reductions are often useful for 2-D visualization of high dimensional data (as shown).



Projection of MNIST dataset using AE
from 784 dimensions to 2

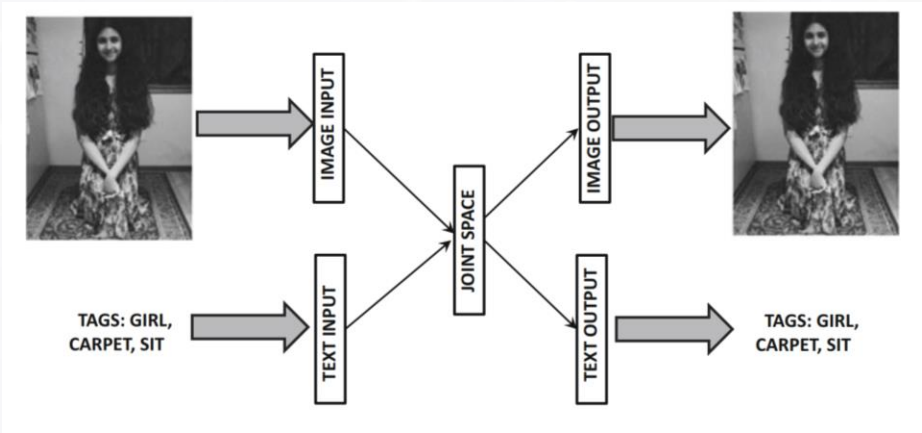
Autoencoders: SVD

• AEs provide further utility as **denoising** workflows.



• AES can also be used for **embedding multimodal data** (where the input data is heterogenous) in a joint latent space.

• In general, multimodal data poses challenges in classical ML settings, because different features often require different types of processing and treatment. By embedding heterogenous attributes in a unified space, one is removing this source of difficulty.



Autoencoders: Example

- Here is a simple, one hidden layer AE applied to MNIST:

```
import keras
from keras import layers

# This is the size of our encoded representations
encoding_dim = 32  # 32 floats -> compression of factor 24.5, assuming the input is 784 floats

# This is our input image
input_img = keras.Input(shape=(784,))
# "encoded" is the encoded representation of the input
encoded = layers.Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = layers.Dense(784, activation='sigmoid')(encoded)

# This model maps an input to its reconstruction
autoencoder = keras.Model(input_img, decoded)

autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

- A deeper AE applied to MNIST:

```
input_img = keras.Input(shape=(784,))
encoded = layers.Dense(128, activation='relu')(input_img)
encoded = layers.Dense(64, activation='relu')(encoded)
encoded = layers.Dense(32, activation='relu')(encoded)

decoded = layers.Dense(64, activation='relu')(encoded)
decoded = layers.Dense(128, activation='relu')(decoded)
decoded = layers.Dense(784, activation='sigmoid')(decoded)
```

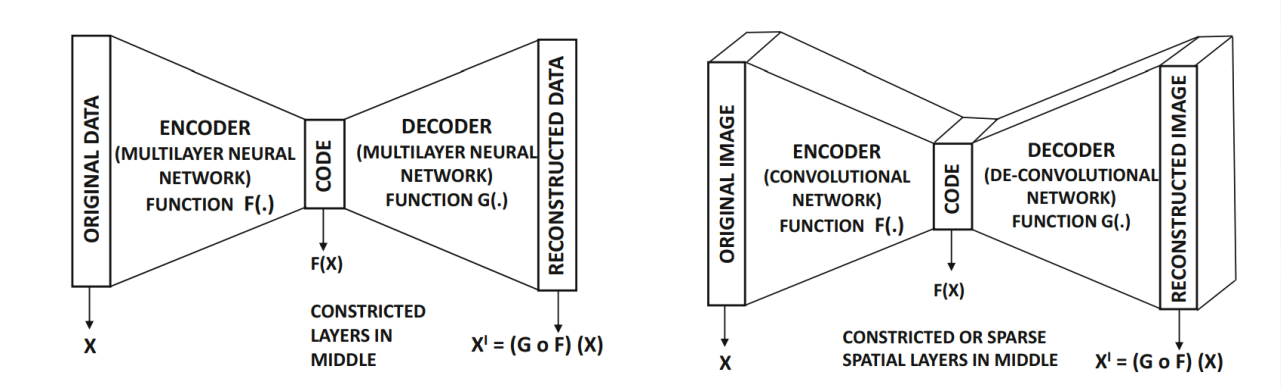
- Denoising AE:

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))

noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)

x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

Convolutional Autoencoders

• A convolutional autoencoder (CAE) functions in the same fashion as an ordinary AE, but with the use of convolutional operations.

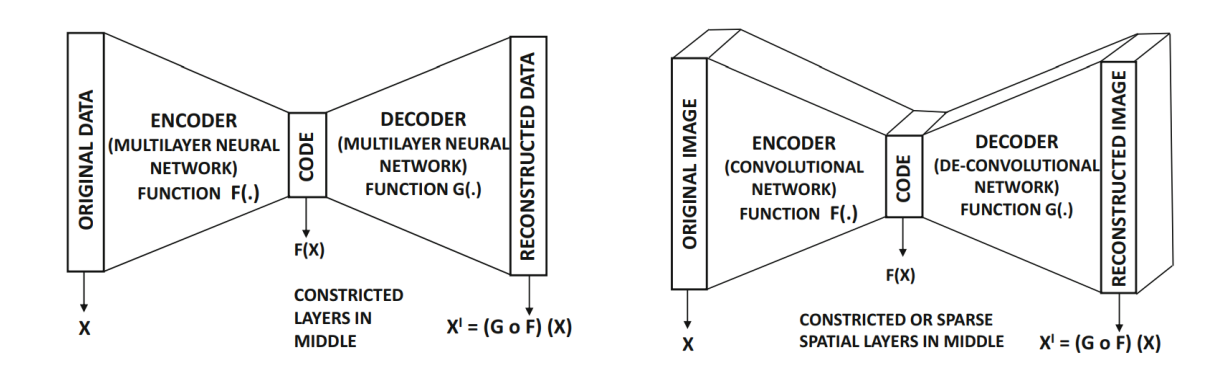
• CAEs introduce a bottleneck module, but they leverage spatial information (via convolution operations), and thus the encoder-decoder subnetworks represent 3D volumes.



Convolutional Autoencoders

• A convolutional autoencoder (CAE) functions in the same fashion as an ordinary AE, but with the use of convolutional operations.

• CAEs introduce a bottleneck module, but they leverage spatial information (via convolution operations), and thus the encoder-decoder subnetworks represent 3D volumes.



• Just as with the compression portion of the encoder of an AE, the encoder in a CAE uses repeated convolutional operations (layer-by-layer) to process a compressed version of the network input.

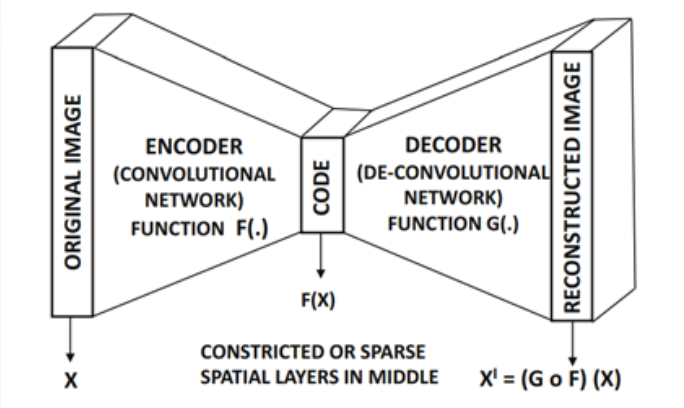
Oftentimes, we use 3D reconstruction error to train the CAE:

$$E = \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{D} \left(X_{ij}^{(k)} - \bar{X}_{ij}^{(k)} \right)$$

Convolutional Autoencoders

• In more detail: there are typically (2) operations we wish to invert in the decoder layers, corresponding to the **convolution** and **max-pooling** and **RELU** of the encoder layers.

(1) The decoder subnetwork consists of **deconvolutional layers** (also called **transposed convolution**), which entails performing matrix multiplication (by a transposed weight matrix) in order to achieve a "deconvolution" of the layer activations. Deconvolution "inverts" convolution operations in the encoder subnetwork, resulting in a "scaling up" of the network feature maps.

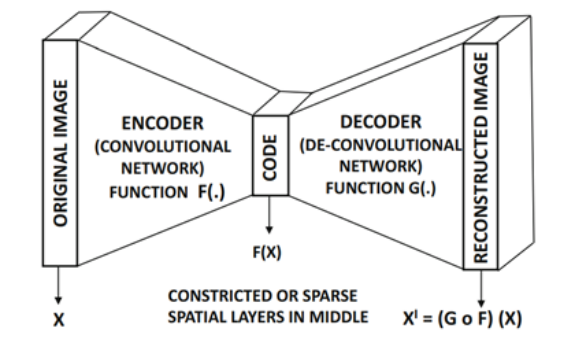


Convolutional Autoencoders

• In more detail: there are (3) operations we wish to invert in the decoder layers, corresponding to the **convolution** and **max-pooling** of the encoder layers.

(1) The decoder subnetwork consists of **deconvolutional layers** (also called **transposed convolution**), which entails performing matrix multiplication (by a transposed weight matrix) in order to achieve a "deconvolution" of the layer activations. Deconvolution "inverts" convolution operations in the encoder subnetwork.

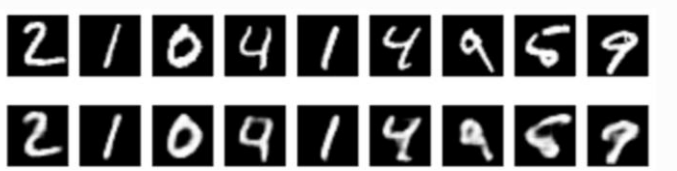
(2) Pooling operations irreversibly lose some information and are therefore impossible to invert exactly. The max-"**unpooling**" operation is implemented with the help of *switches*. Typically, with unpooling, the feature map is increase by a factor (e.g., of 2), and the values stored at the "switch" position from the corresponding maxpooling in the encoder layer is copied from the previous layer (the other values are set to zero).



Convolutional Autoencoders

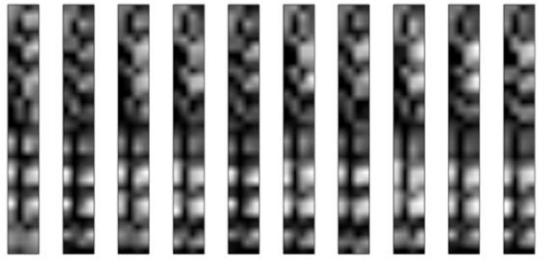
•Here is a coded exampled of a CAE with 3 conv layers, with each followed by pooling and RELU*:

```
input_img = keras.Input(shape=(28, 28, 1))

x = layers.Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = layers.MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (4, 4, 8) i.e. 128-dimensional

x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(16, (3, 3), activation='relu')(x)
x = layers.UpSampling2D((2, 2))(x)
decoded = layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = keras.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```



Visualizing the 128-dimensional encodings:

```
encoder = keras.Model(input_img, encoded)
encoded_imgs = encoder.predict(x_test)

n = 10
plt.figure(figsize=(20, 8))
for i in range(1, n + 1):
    ax = plt.subplot(1, n, i)
    plt.imshow(encoded_imgs[i].reshape((4, 4 * 8)).T)
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



*This implementation uses conv-upsampling blocks to achieve an inversion of conv-maxpooling in the encoder network. This is also a common implementation schematic for CAEs.

Visualization

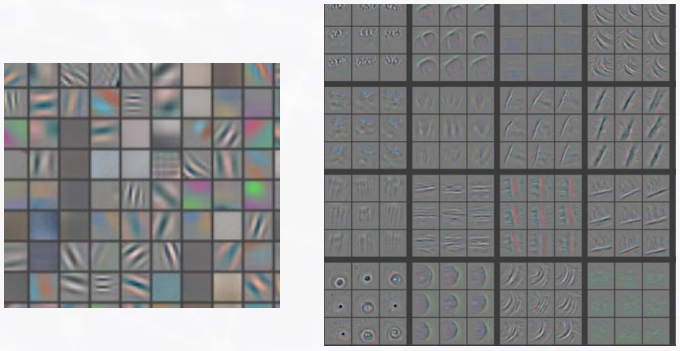
• Unlike generic DNNs, **CNNs are highly interpretable models** due to the spatial nature of learning via convolutional operations.

• In the most straightforward sense, one can simply visualize the 2D (spatial) components of the learned filters in a CNN (see below).

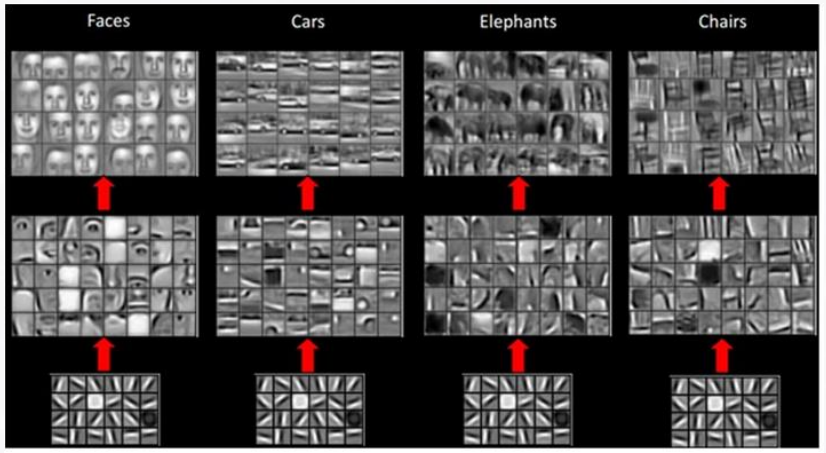
Visualization

• Unlike generic DNNs, **CNNs are highly interpretable models** due to the spatial nature of learning via convolutional operations.

• In the most straightforward sense, one can simply visualize the 2D (spatial) components of the learned filters in a CNN (see below).



First (left) & second (right) layer filters in a CNN



• Although this type of visualization can provide some interesting instances of the primitive edges and related features learned in the first layer of the CNN, it is not very useful for later layers.

• In the **first layer**, it is possible to visualize these filters directly because they operate directly on the input image. For later layers, convolutions are aggregated across layers, leading to increasingly abstract learned filters in the later layers of a CNN.

Visualization

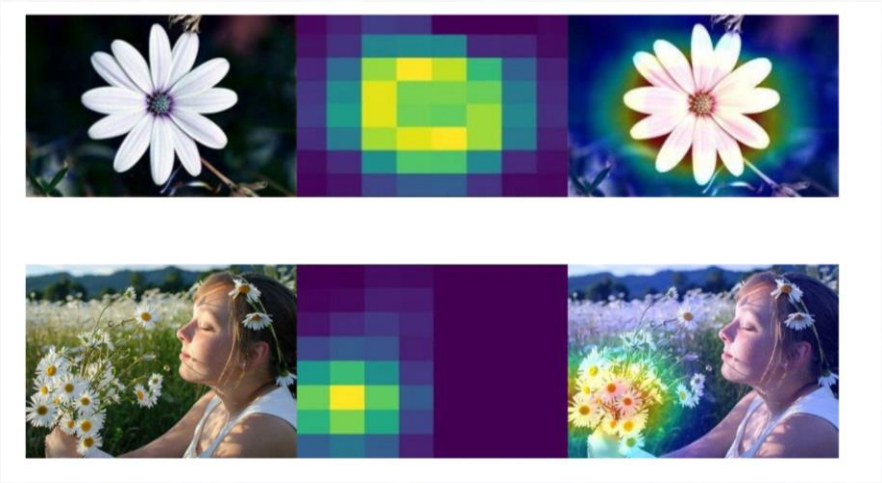
• In order to obtain a more coherent form of interpretability, one must find a way to map the impacts of all operations back to the input layer of the network.

• Therefore, the goal of network visualization is often to identify and highlight portions of the input image to which a particular hidden feature is responding.

Visualization

• In order to obtain a more coherent form of interpretability, one must find a way to map the impacts of all operations back to the input layer of the network.

• Therefore, the goal of network visualization is often to identify and highlight portions of the input image to which a particular hidden feature is responding.

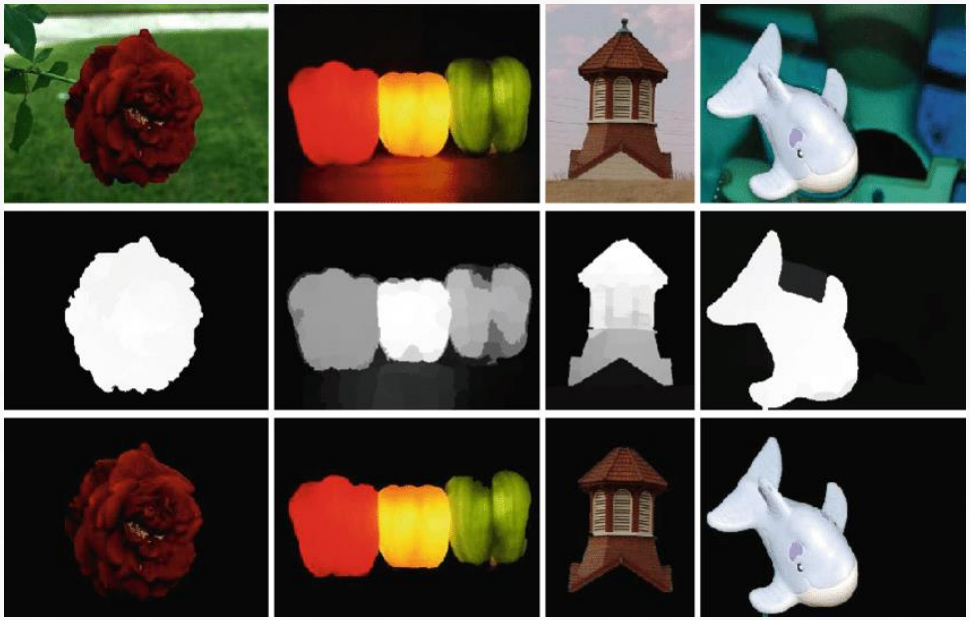


•For instance, the value of a hidden feature might be sensitive to changes in the portion of the image corresponding to a face, a flower, or a traffic sign, etc. This is naturally achieved by computing the sensitivity (via the gradient) of a hidden feature with respect to each pixel of the input image.

Visualization: Saliency

• In CV, a **saliency map** is an image that shows each pixel's unique quality. The goal of a saliency map is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze for a CV workflow.

• Saliency is closely related to **image segmentation** (a topic we study in more detail later in our course); in fact, saliency detection is frequently used as a pre-step for object segmentation in CV.

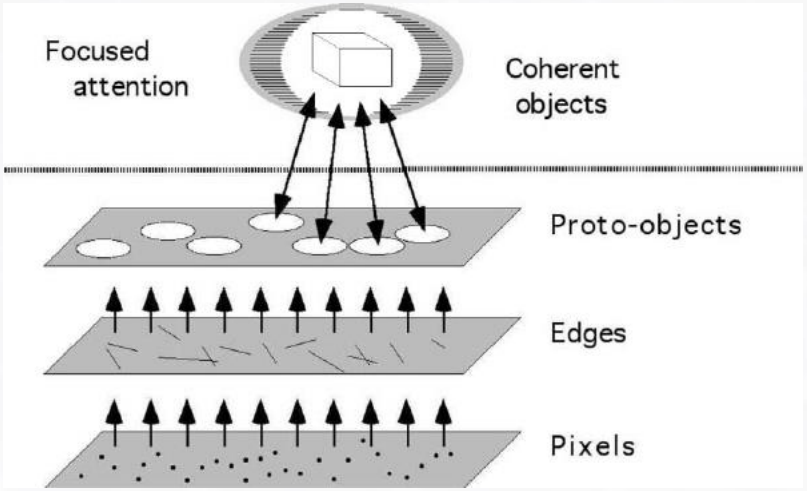


Visualization: Saliency

• In the visual system, it is widely believed that two basic, complementary stages of visual processing take place*:

(1) The **pre-attentive process** where low-level features such as orientation, edges, or intensities can "pop up" automatically. From the perspective of object detection, pre-attentive objects become candidates for object detection. These pre-attentive objects are sometimes called *proto objects*.

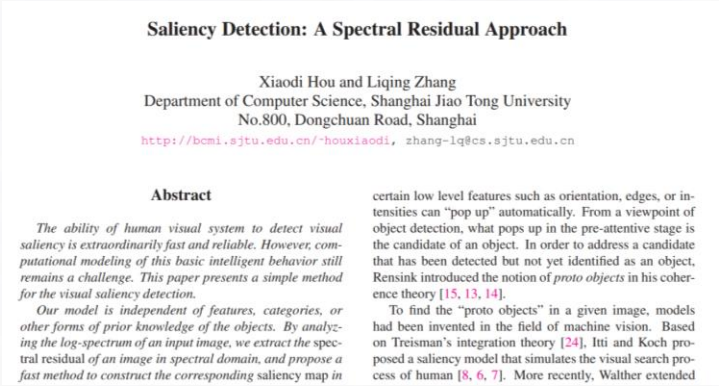
(2) An **attention process** that is generally slower, executed in a serial fashion, and compute-intensive relative to the pre-attentive process.



*https://www.sciencedirect.com/science/article/pii/S0042698900000031

Visualization: Classical Saliency

• For an introduction to classical saliency algorithms, we now review **spectral residual saliency*** (Hou *et al.*, 2007) a component of OpenCV library.



Saliency Detection: A Spectral Residual Approach

Xiaodi Hou and Liqing Zhang
Department of Computer Science, Shanghai Jiao Tong University
No.800, Dongchuan Road, Shanghai
http://bcmi.sjtu.edu.cn/~houxiaodi, zhang-lq@cs.sjtu.edu.cn

Abstract

The ability of human visual system to detect visual saliency is extraordinarily fast and reliable. However, computational modeling of this basic intelligent behavior still remains a challenge. This paper presents a simple method for the visual saliency detection.

Our model is independent of features, categories, or other forms of prior knowledge of the objects. By analyzing the log-spectrum of an input image, we extract the spectral residual of an image in spectral domain, and propose a fast method to construct the corresponding saliency map in

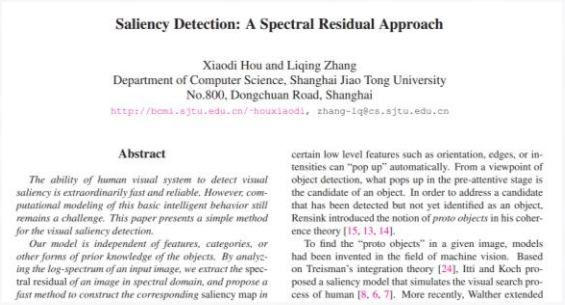
certain low level features such as orientation, edges, or intensities can "pop up" automatically. From a viewpoint of object detection, what pops up in the pre-attentive stage is the candidate of an object. In order to address a candidate that has been detected but not yet identified as an object, Rensink introduced the notion of *proto objects* in his coherence theory [15, 13, 14].

To find the "proto objects" in a given image, models had been invented in the field of machine vision. Based on Treisman's integration theory [24], Itti and Koch proposed a saliency model that simulates the visual search process of human [8, 6, 7]. More recently, Walther extended

• It is well-known that natural images exhibit a general, **scale invariant property**. In other words, the distribution of image statistics for natural images remains unchanged if the image is scaled.

Visualization: Classical Saliency

• For an introduction to classical saliency algorithms, we review **spectral residual saliency*** (Hou *et al.*, 2007) a component of OpenCV library.



Saliency Detection: A Spectral Residual Approach

Xiaodi Hou and Liqing Zhang
Department of Computer Science, Shanghai Jiao Tong University
No.800, Dongchuan Road, Shanghai
http://bcmi.sjtu.edu.cn/~houxiaodi, zhang-lq@cs.sjtu.edu.cn

Abstract

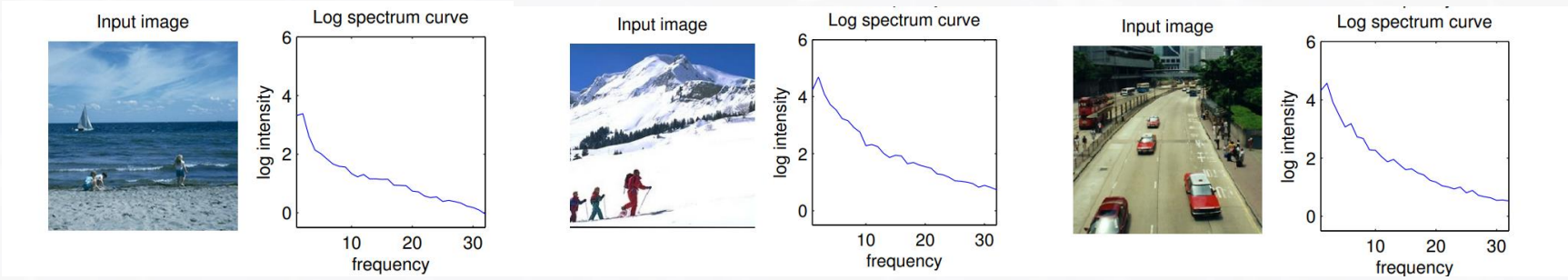
The ability of human visual system to detect visual saliency is extraordinarily fast and reliable. However, computational modeling of this basic intelligent behavior still remains a challenge. This paper presents a simple method for the visual saliency detection.

Our model is independent of features, categories, or other forms of prior knowledge of the objects. By analyzing the log-spectrum of an input image, we extract the spectral residual of an image in spectral domain, and propose a fast method to construct the corresponding saliency map *in* certain low level features such as orientation, edges, or intensities can "pop up" automatically. From a viewpoint of object detection, what pops up in the pre-attentive stage is the candidate of an object. In order to address a candidate that has been detected but not yet identified as an object, Rensink introduced the notion of *proto objects* in his coherence theory [15, 13, 14].

To find the "proto objects" in a given image, models had been invented in the field of machine vision. Based on Treisman's integration theory [24], Itti and Koch proposed a saliency model that simulates the visual search process of human [8, 6, 7]. More recently, Walther extended

• It is well-known that natural images exhibit a general, **scale invariant property**. In other words, the distribution of image statistics for natural images remains unchanged if the image is scaled.

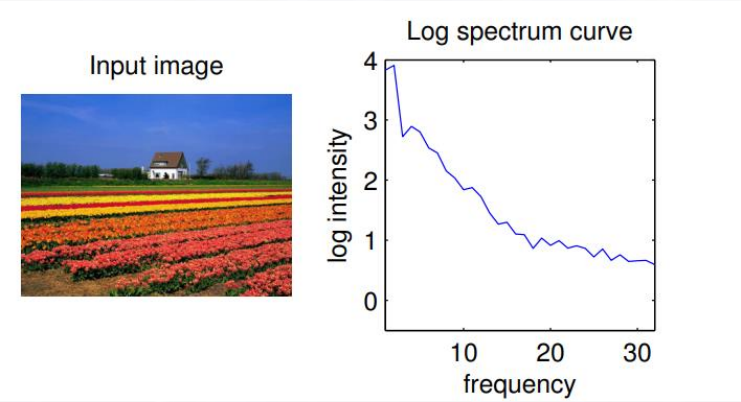
• This property is encapsulated by the **power law**, which is to say the amplitude A(f) of the averaged Fourier spectrum of an ensemble of natural images obeys the distribution: $E\{L(f)\} \propto 1/f$



*http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.125.5641&rep=rep1&type=pdf

Visualization: Classical Saliency

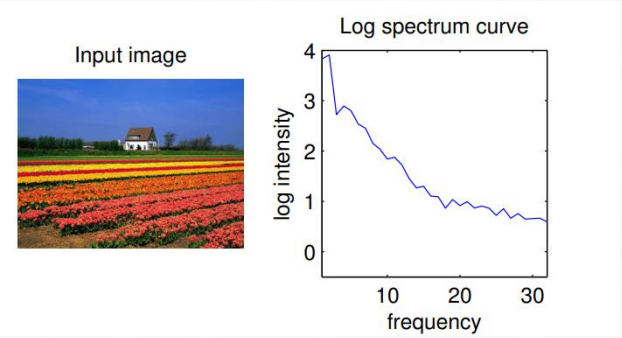
- Given an input image, the authors first compute the **log of the Fourier transform** of the image: $L(f)$.



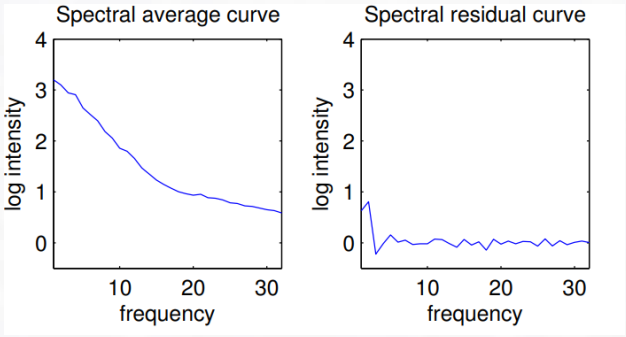
$L(f)$

Visualization: Classical Saliency

• Given an input image, the authors first compute the **log of the Fourier transform** of the image: $L(f)$.



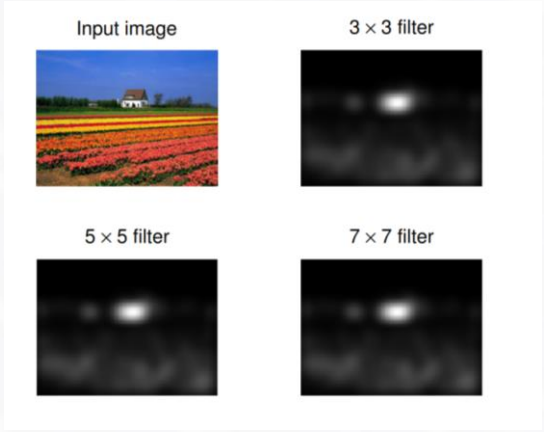
$L(f)$



$A(f) = h_n(f) * L(f)$ $\quad R(f) = L(f) - A(f)$

• Because the log transform of a natural image is locally linear (see image above), the average log spectrum can be approximated using a <u>local average filter</u> over an $n \times n$ window:

$$h_n(f) = \frac{1}{n^2}\begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}$$



• The **averaged spectrum** can then be approximated by convolving:

$$A(f) = h_n(f) * L(f)$$

• The authors define the **spectral residual** R(f): $R(f) = L(f) - A(f)$

Visualization: Classical Saliency

$$F(I) \rightarrow L(f) \rightarrow \underbrace{A(f) = h_n(f) * L(f)}_{\text{Average spectrum approximation}} \rightarrow \underbrace{R(f) = L(f) - A(f)}_{\text{spectral residual}}$$

Fourier Transform of Input Image *I*

log Fourier

• To generate a **saliency map**, S(x), the authors then perform an inverse Fourier transform (F^{-1}) on the spectral residual, followed by Gaussian smoothing:

Visualization: Classical Saliency

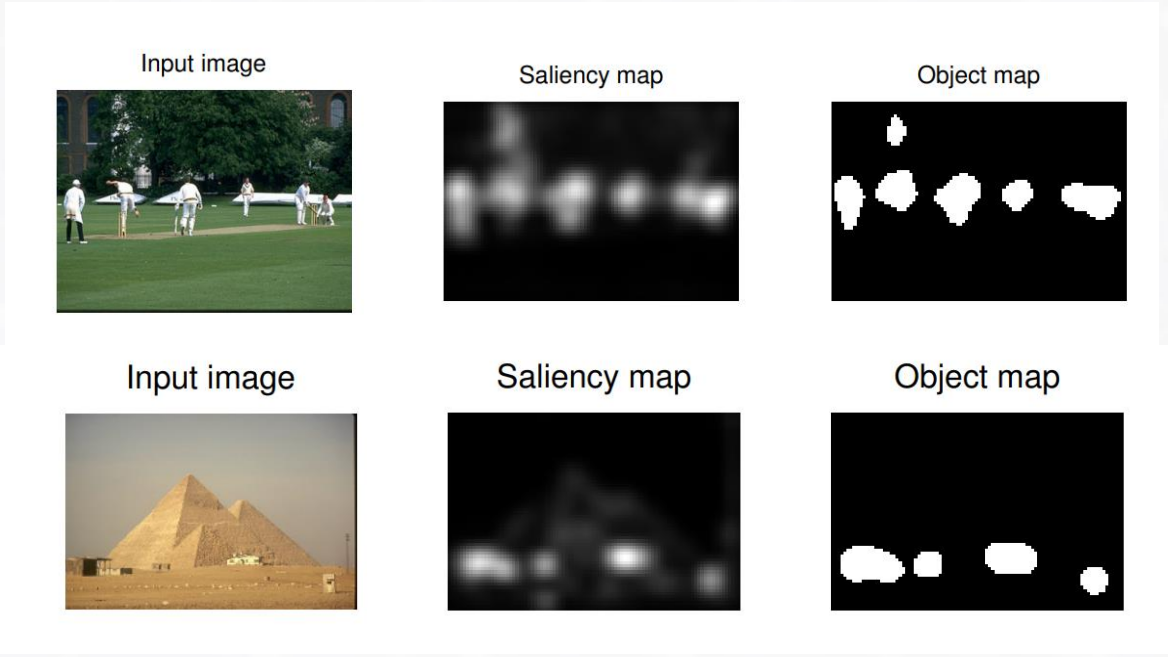
$$F(I) \rightarrow L(f) \rightarrow \underbrace{A(f) = h_n(f) * L(f)}_{\text{Average spectrum approximation}} \rightarrow \underbrace{R(f) = L(f) - A(f)}_{\text{spectral residual}}$$

Fourier Transform of Input Image *I*

log Fourier

- To generate a **saliency map**, $S(x)$, the authors then perform an inverse Fourier transform (F^{-1}) on the spectral residual, followed by Gaussian smoothing:

- $S(x)$ is then thresholded to (i.e., $S(x) > \delta$) for object detection.



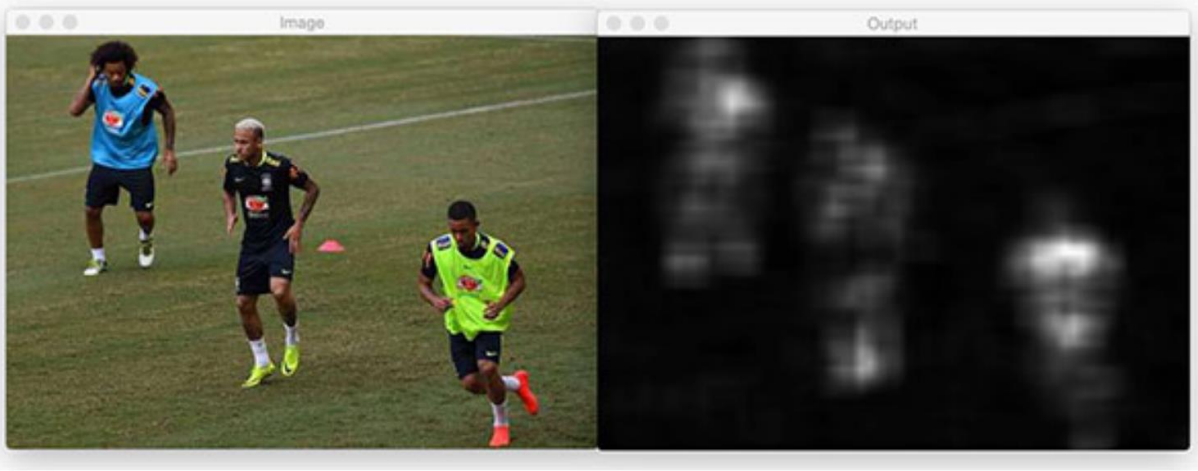
Visualization: Classical Saliency

$$F(I) \rightarrow L(f) \rightarrow \underbrace{A(f) = h_n(f) * L(f)}_{\text{Average spectrum approximation}} \rightarrow \underbrace{R(f) = L(f) - A(f)}_{\text{spectral residual}}$$

Fourier Transform of Input Image *I*

log Fourier

```
# initialize OpenCV's static saliency spectral residual detector and
# compute the saliency map
saliency = cv2.saliency.StaticSaliencySpectralResidual_create()
(success, saliencyMap) = saliency.computeSaliency(image)
saliencyMap = (saliencyMap * 255).astype("uint8")
cv2.imshow("Image", image)
cv2.imshow("Output", saliencyMap)
cv2.waitKey(0)
```



Visualization

• Consider a NN that has been trained using a large dataset like *ImageNet*. The goal is to visualize and understand the impact of the different portions of the input image on various features in the hidden layers and the output layers.

In general, to this end, we would like to address the following:

Visualization

• Consider a NN that has been trained using a large dataset like *ImageNet*. The goal is to visualize and understand the impact of the different portions of the input image on various features in the hidden layers and the output layers.

In general, we would like to address the following:

(1) Given an activation of a feature anywhere in the NN for a particular input image, visualize the portions of the input to which that feature is responding the most.

• The feature might be from a convolutional layer, fully-connected layer, or even the output layer. Note that in the latter case, for example, if an input image is activating the label for "dog", we hope to see parts of the specific input image that look most like a dog.

Visualization

• Consider a NN that has been trained using a large dataset like *ImageNet*. The goal is to visualize and understand the impact of the different portions of the input image on various features in the hidden layers and the output layers.

In general, we would like to address the following:

(1) Given an activation of a feature anywhere in the NN for a particular input image, visualize the portions of the input to which that feature is responding the most.

• The feature might be from a convolutional layer, fully-connected layer, or even the output layer. Note that in the latter case, for example, if an input image is activating the label for "dog", we hope to see parts of the specific input image that look most like a dog.

(2) Given a particular feature anywhere in the NN, render a "dream image" that is likely to activate that feature the most. This is sometimes called the **image inversion problem***. Again, the feature could come from any convolutional layer, FC layer or output layer.

*https://arxiv.org/pdf/1611.03679.pdf

Visualization

Gradient-Based Visualization of Activated Features

• While backpropagation is conventionally used to train a CNN, it can also be used to generate gradient-based visualization.

• The main difference from the traditional backpropagation setting is that our end-goal is to determine the sensitivity of the hidden/output features with respect to different pixels of the input image rather than with respect to the weights. However, even traditional backpropagation does compute the sensitivity of the outputs with respect to various layers as an intermediate step.

Visualization

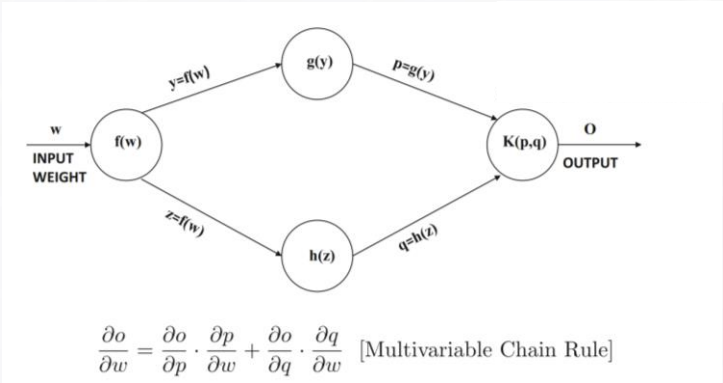
Gradient-Based Visualization of Activated Features

• While backpropagation is conventionally used to train a CNN, it can also be used to generate gradient-based visualization.

• The main difference from the traditional backpropagation setting is that our end-goal is to determine the sensitivity of the hidden/output features with respect to different pixels of the input image rather than with respect to the weights. However, even traditional backpropagation does compute the sensitivity of the outputs with respect to various layers as an intermediate step.

• When the sensitivity of an output o is computed with respect to the input pixels, the visualization of this sensitivity over the corresponding pixels is referred to as a **saliency map**.

• For example, the output o might be the softmax probability of the class "flower". Then, for each pixel xi x_i in the image we want to determine the value of $\frac{\partial o}{\partial x_i}$. Simply, this value can be computed by straightforward* backpropagation to the input layer.



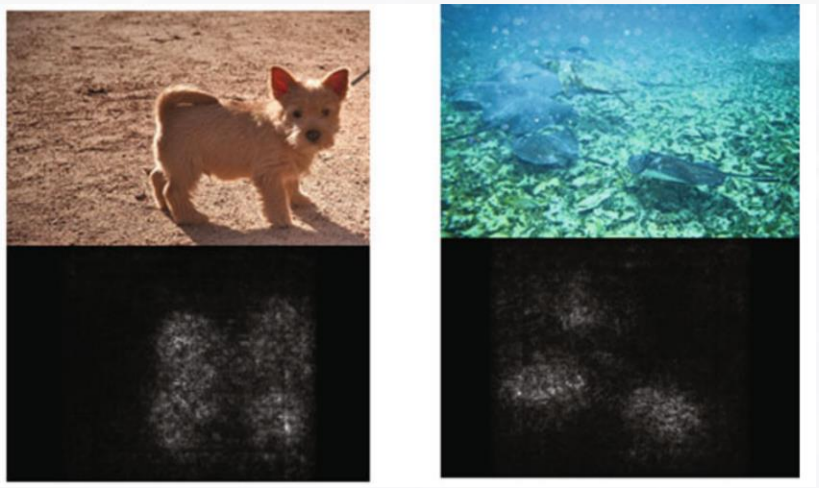
Visualization

Gradient-Based Visualization of Activated Features

• The softmax probability of "flower" will be relatively insensitive to small changes in portions of the image that are irrelevant to the recognition of flower. Therefore, the values $\frac{\partial o}{\partial x_i}$ will be close to zero for such irrelevant regions.

• In more detail: Consider a pre-trained AlexNet with input image of dimension 224x224x3. We compute $\frac{\partial o}{\partial x_i}$ over this volume, generating a gradient tensor of the same dimension.

• To aid in visualization, we convert this gradient volume into a grayscale image by calculating the maximum of the absolute magnitude of the gradient over the three RGB channels, yielding a 224x224x1 **saliency map** with non-negative values.



https://arxiv.org/pdf/1312.6034.pdf

Visualization

Gradient-Based Visualization of Activated Features

• This general approach has also been used for visualizing the activations of specific hidden features. Consider the value h of a hidden variable (i.e., a neuron in a CNN) for a particular input image. How is this variable responding to the input image at its current activation level?

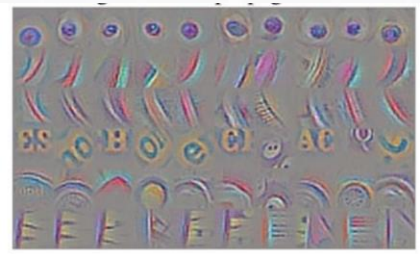
• The idea is that if we slightly increase or decrease the color intensity of some pixels, the value h will be affected more than if we increase/decrease other pixels. In addition, the **receptive field** of h will depend on its placement in the network. For neurons in the first layer (say of VGG), h will have a receptive field of size 3x3; conversely, neurons in later layers in the network will have a larger receptive field.

Visualization

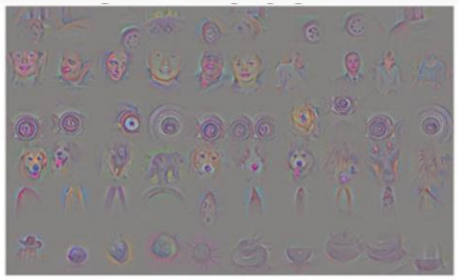
Gradient-Based Visualization of Activated Features

• This general approach has also been used for visualizing the activations of specific hidden features. Consider the value h of a hidden variable (i.e., a neuron in a CNN) for a particular input image. How is this variable responding to the input image at its current activation level?

• The idea is that if we slightly increase or decrease the color intensity of some pixels, the value h will be affected more than if we increase/decrease other pixels. In addition, the **receptive field** of h will depend on its placement in the network. For neurons in the first layer (say of VGG), h will have a receptive field of size 3x3; conversely, neurons in later layers in the network will have a larger receptive field.



(Left) Layer 6-based gradient $\left(\frac{\partial h}{\partial x_i}\right)$ for corresponding image crops (Right)



(Left) Layer 9-based gradient $\left(\frac{\partial h}{\partial x_i}\right)$ for corresponding image crops (Right)

• At a high level of activation of h, some of the pixels in that receptive field will be more sensitive to h than others. By isolating the pixels to which the hidden variable h has the greatest sensitivity and visualizing the corresponding regions, one can get an idea of what part of the input most affects a particular hidden feature.

Visualization

Gradient-Based Visualization of Activated Features

• The previous discussion focused on visualizing elements of the input image to which a feature (i.e., output neuron, hidden layer neuron, etc.) is responding the most.

• Remarkably, we can invert the visualization process to instead render a synthesized image that activates a feature (including output class)!

Visualization

Gradient-Based Visualization of Activated Features

• For simplicity, consider o to be the unnormalized score for a particular output image class, e.g., "shiba inu." We would like to learn the input image $\bar{x}$ that maximizes the output o, while applying some regularization to $\bar{x}$ (we apply the regularization so that the output image conforms with basic semantic properties of images* – this is sometimes called an **image prior**):

$$\arg\max_{\bar{x}} J(\bar{x}) = (o - \lambda \|\bar{x}\|^2)$$

where λ is the regularization parameter and is important to extract semantically interpretable images; this particular form of regularization clips the image intensities.

*Oftentimes intensity clipping, and minimization of **total variation of an image**, are imposed as image prior conditions; see: https://arxiv.org/pdf/1412.0035.pdf

Visualization

Gradient-Based Visualization of Activated Features

• For simplicity, consider o to be the unnormalized score for a particular output image class, e.g., "shiba inu." We would like to learn the input image $\bar{x}$ that maximizes the output o, while applying some regularization to $\bar{x}$ (we apply the regularization so that the output image conforms with basic semantic properties of images* – this is sometimes called an **image prior**):

$$\arg\max_{\bar{x}} J(\bar{x}) = (o - \lambda \|\bar{x}\|^2)$$

where λ is the regularization parameter and is important to extract semantically interpretable images; this particular form of regularization clips the image intensities.

*Oftentimes intensity clipping, and minimization of **total variation of an image**, are imposed as image prior conditions; see: https://arxiv.org/pdf/1412.0035.pdf

In literature this inversion problem is often generally framed as: $\bar{x} = \arg\min_{\mathbf{x} \in \mathbb{R}^{H \times W \times C}} \left[l\left(\Phi(\mathbf{x}), \Phi_0\right) + \lambda \underbrace{R(\mathbf{x})}_{\text{Regularizer}} \right]$

where **x** denotes the inverse image, $\Phi(\cdot)$ a representation function (e.g., feature map of a hidden layer in a NN); $\Phi_0 = \Phi(x_0)$ is the reference image representation.

Visualization

Gradient-Based Visualization of Activated Features

$$\arg\max_{\bar{x}} J(\bar{x}) = (o - \lambda \|\bar{x}\|^2)$$

• One can approximate a solution to this optimization problem using **gradient ascent** in conjunction with backpropagation in order to learn the image $\bar{x}$ maximizing the objective function.

Visualization

Gradient-Based Visualization of Activated Features

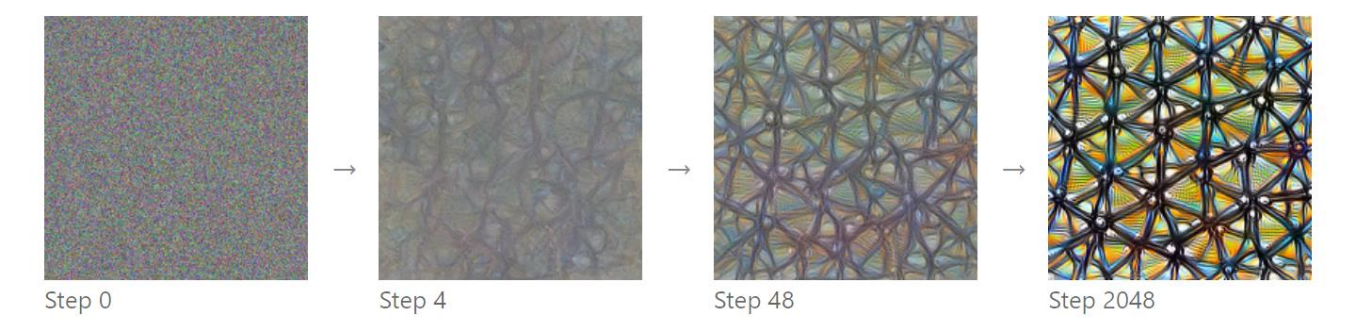
$$\arg\max_{\bar{x}} J(\bar{x}) = (o - \lambda \|\bar{x}\|^2)$$

• One can approximate a solution to this optimization problem using **gradient ascent** in conjunction with backpropagation in order to learn the image $\bar{x}$ maximizing the objective function.

Thus, we start with a zero image $\bar{x}_0$ (or random noise) and update $\bar{x}_{i+1}$ using gradient ascent:

$$\bar{x}_{i+1} \leftarrow \bar{x}_i + \alpha \nabla_{\bar{x}} J(\bar{x})$$

• Here α is the learning rate. The key point is that backpropagation is being leveraged in an unusual way to update *image pixels* while keeping the (already learned) weights of the model fixed.

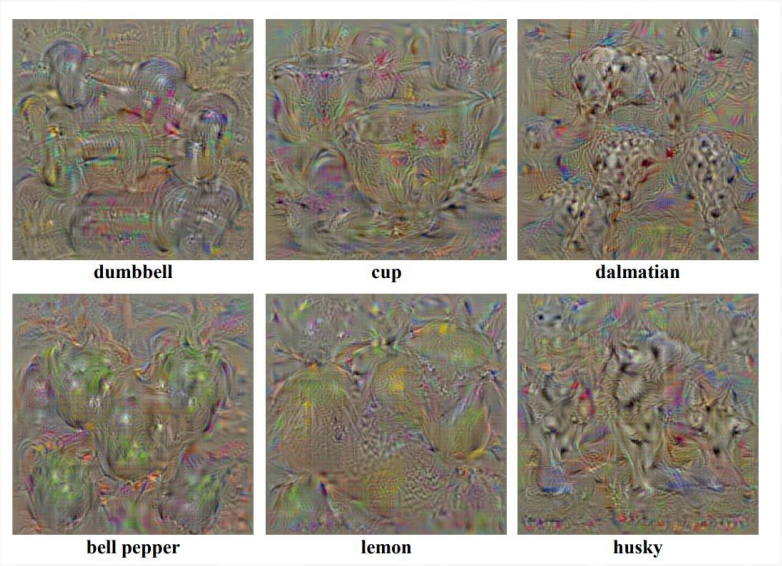


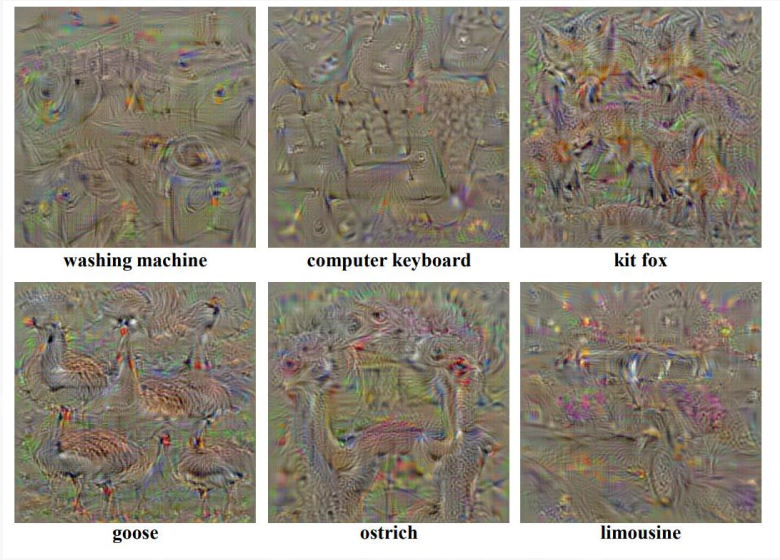
Visualization

Gradient-Based Visualization of Activated Features

$$\arg\max_{\bar{x}} J(\bar{x}) = (o - \lambda \|\bar{x}\|^2)$$

$$\bar{x}_{i+1} \leftarrow \bar{x}_i + \alpha \nabla_{\bar{x}} J(\bar{x})$$





• Above are examples of synthesized images from "Deep Inside Convolutional Networks: Visualizing Image Classification Models and Saliency Maps" (*Simonyan et al., 2014).

• These images are illustrative of what the trained network perceives – at least loosely – to constitute "idealized" instantiations of a particular category.

*https://arxiv.org/pdf/1312.6034.pdf

Visualization

Gradient-Based Visualization of Activated Features

• An analogous methodology can be applied to actualize synthetic images that activate hidden layer neurons h (in lieu of output neurons o). Happily, the methodology is essentially no different, as we once again use gradient ascent:

$$\arg\max_{\bar{x}} J(\bar{x}) = (h - \lambda \|\bar{x}\|^2)$$

$$\bar{x}_{i+1} \leftarrow \bar{x}_i + \alpha \nabla_{\bar{x}} J(\bar{x})$$

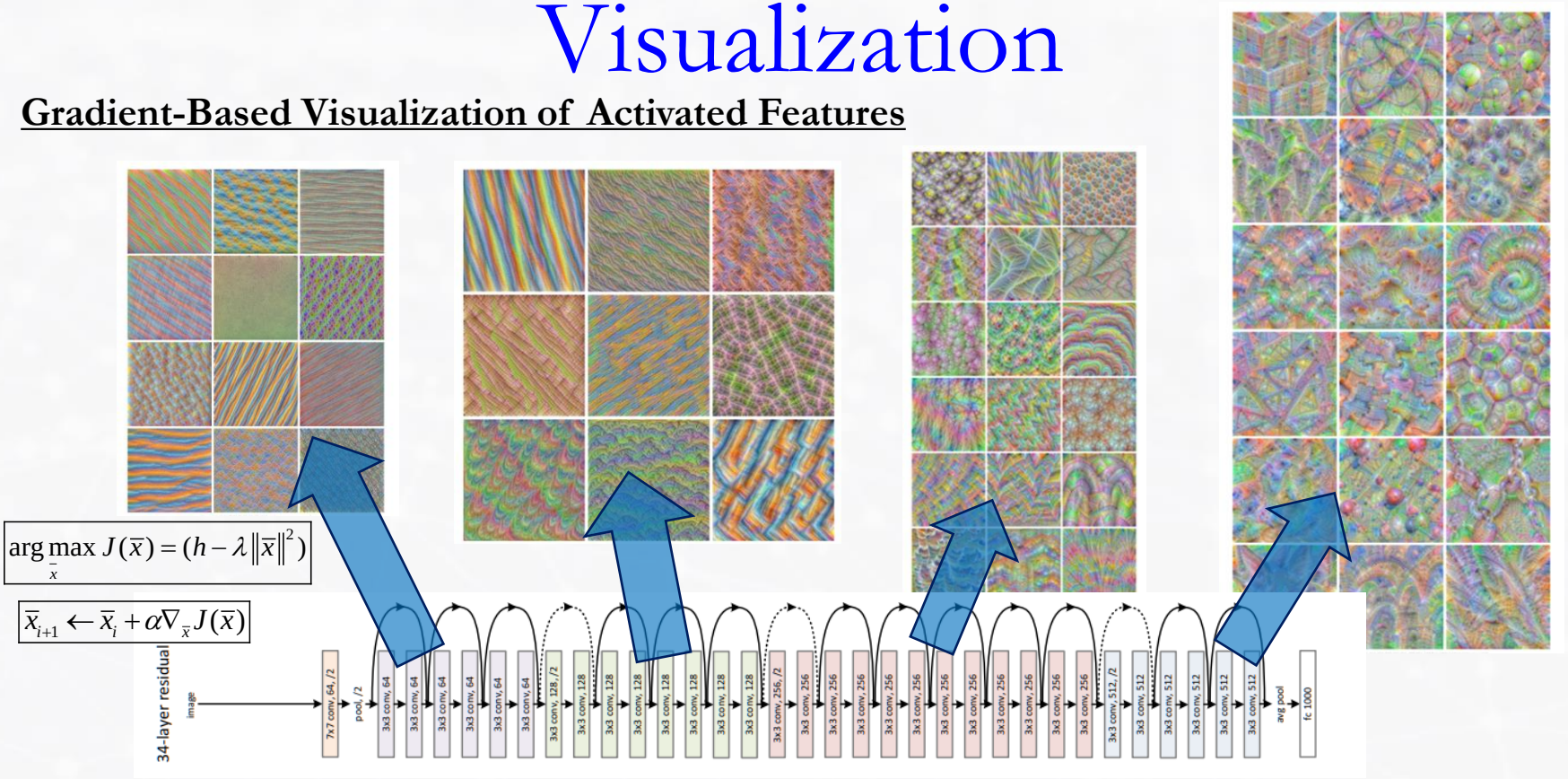
• Visualizing synthetic images in this fashion enlightens us about several intriguing aspects of CNNs, including:

(1) The **individual learned features** – and their relation to other learned features – (including high-level features) in the network.

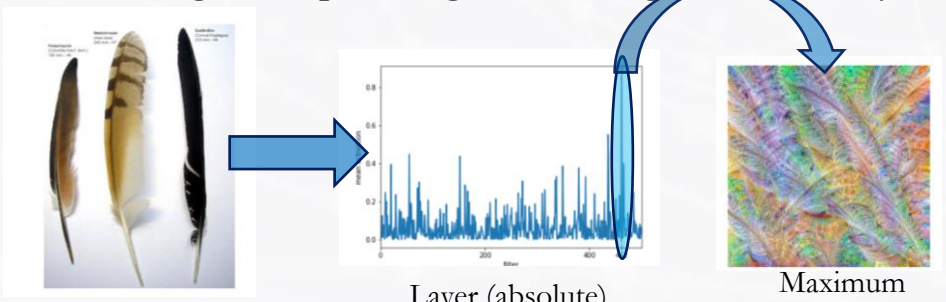
(2) The **effect and function of the receptive field** of the network components.

Visualization

Gradient-Based Visualization of Activated Features



• Above, visualizations from ResNet-34 hidden neurons. Oftentimes, the feature visualizations appear as gossamer composites; (below) the absolute activation for a given input image, indicating the sensitivity to the class of "feathers" in this case.



Input image

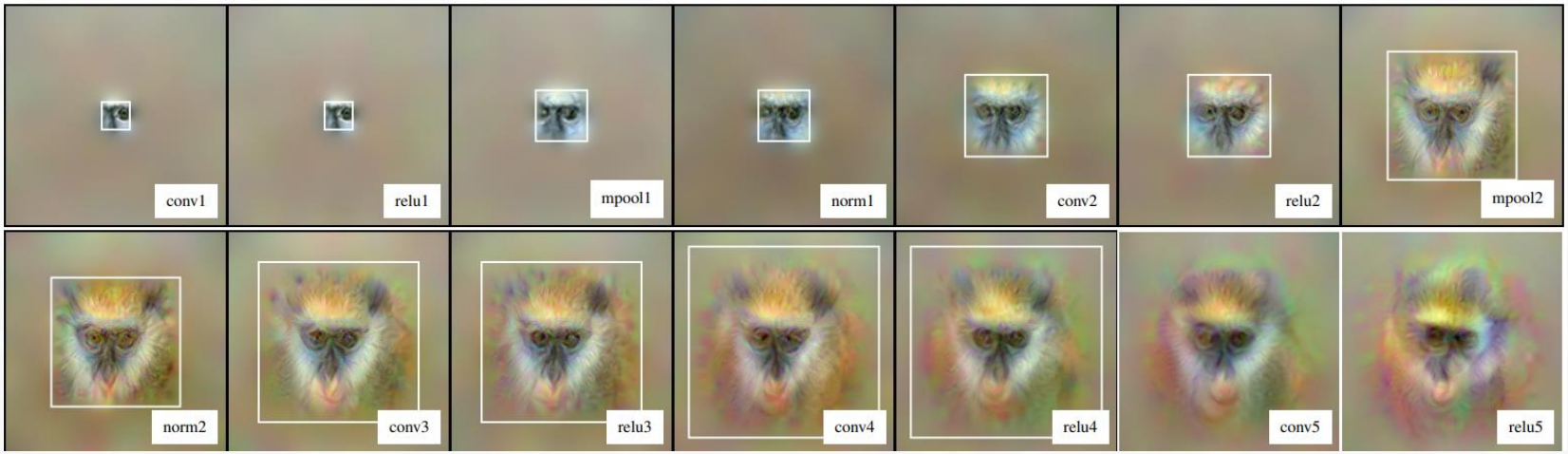
Layer (absolute) filter activations

Maximum activation filter

https://towardsdatascience.com/how-to-visualize-convolutional-features-in-40-lines-of-code-70b7d87b0030

Visualization

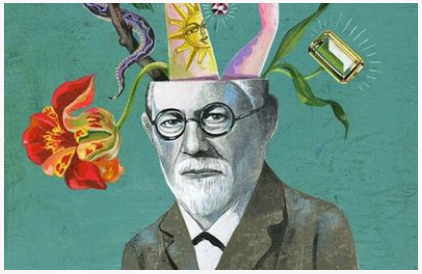
Gradient-Based Visualization of Activated Features



• Above we see the difference in the receptive field of different neurons in deeper layers of a trained CNN.

Visualization: Dream Deeper

• Following the publication of several influential papers on CNNs and deep model visualization (including the work we have reviewed here), Google created **DeepDream** (2015), a program that creates dream-like hallucinogenic images from pre-trained CNNs.



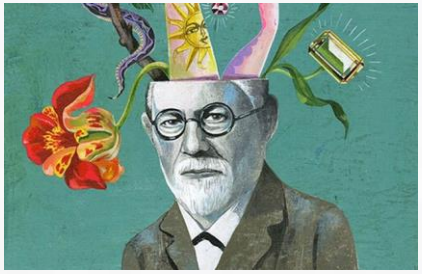
*"The interpretation of dreams is the royal road to a knowledge of the unconscious activities of the mind."**
-- Freud

• DeepDream uses a pre-trained CNN (e.g., Inception) and applies *gradient ascent* to the activations of a set of layers in the network in response to an input image (these layers can be chosen at random, or pre-selected).

*https://www.freud.org.uk/education/resources/the-interpretation-of-dreams/

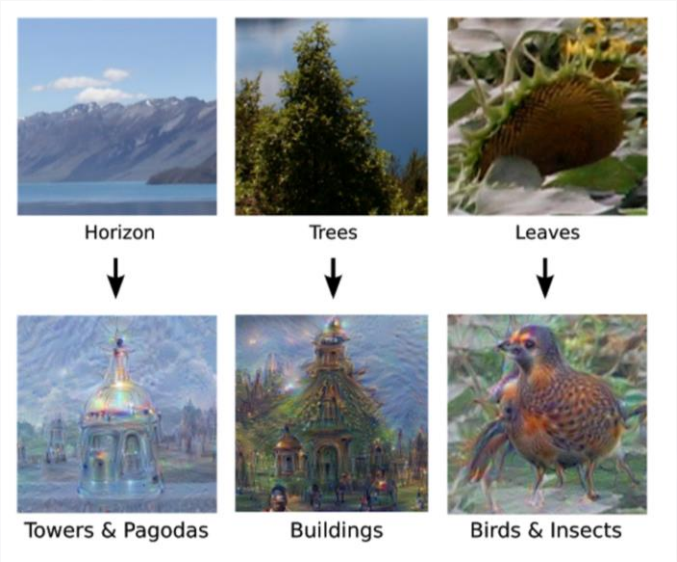
Visualization: Dream Deeper

• Following the publication of several influential papers on CNNs and deep model visualization (including the work we have reviewed here), Google created **DeepDream** (2015), a program that creates dream-like hallucinogenic images from pre-trained CNNs.



• DeepDream uses a pre-trained CNN (e.g., Inception) and applies *gradient ascent* to the activations of set of layers in the network in response to an input image (these layers can be chosen at random, or pre-selected).

• Notice that the **choice of layers** in addition to the **input image** (naturally) will have a significant impact on the resulting image, due to the sensitivity of each neuron to specific image/class features.

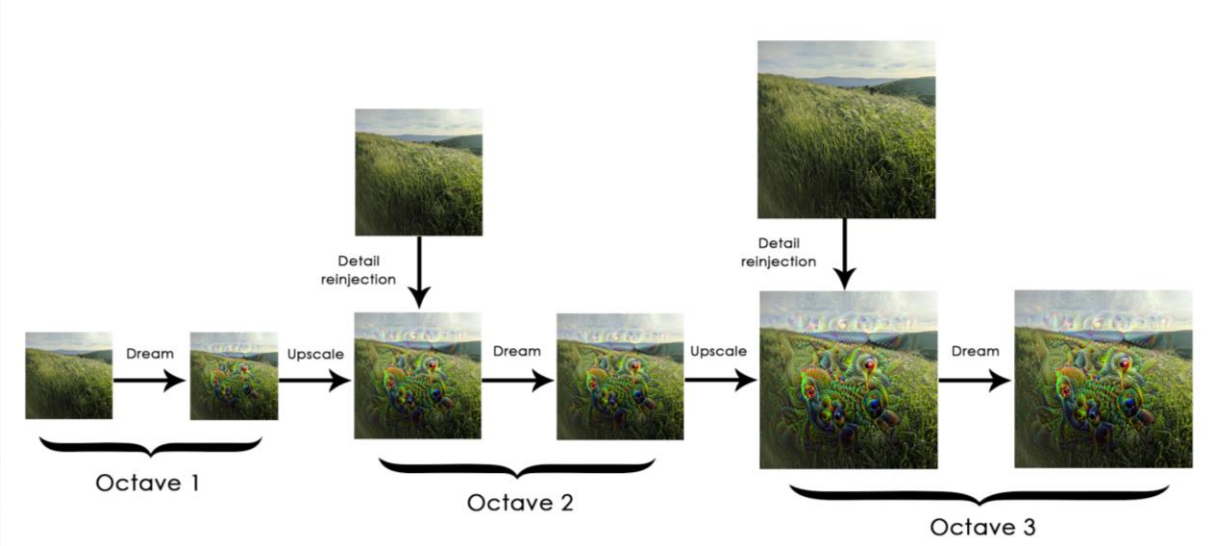


• The results vary quite a bit with the kind of image, because the **features that are entered bias the network** towards certain interpretations. For example, horizontal lines tend to get filled with towers and pagodas. Rocks and trees turn into buildings. Birds and insects appear in images of leaves.

• This technique gives us a **qualitative sense** of the level of abstraction that a particular layer has achieved in its understanding of images.

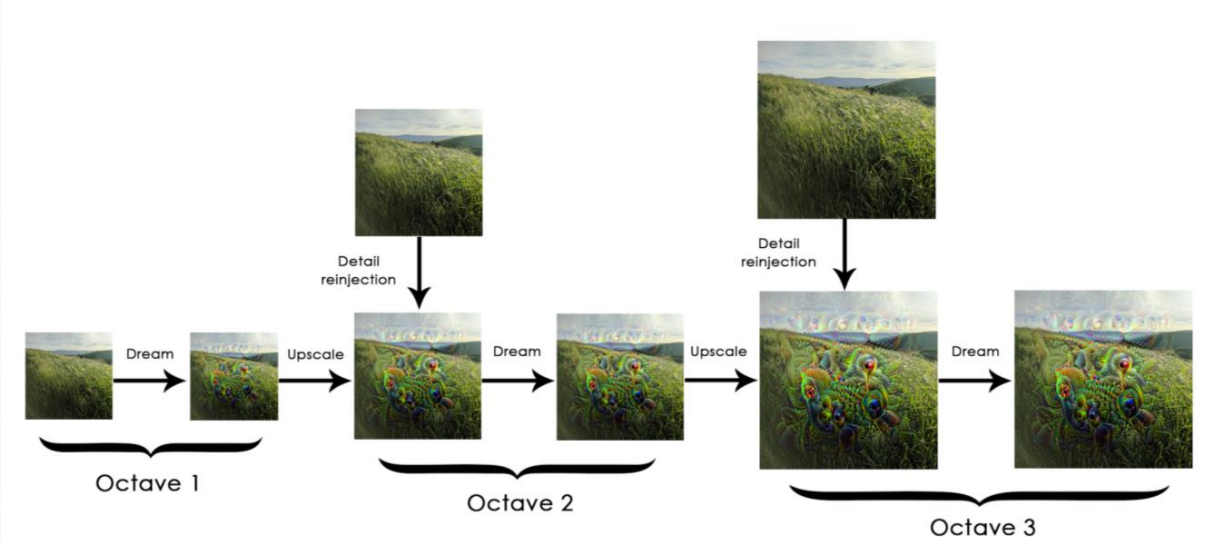
https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html

Visualization: Dream Deeper



• The original image is processed at different scales "octaves" (typically a small number, e.g., 3, ~40% rescale size).

Visualization: Dream Deeper

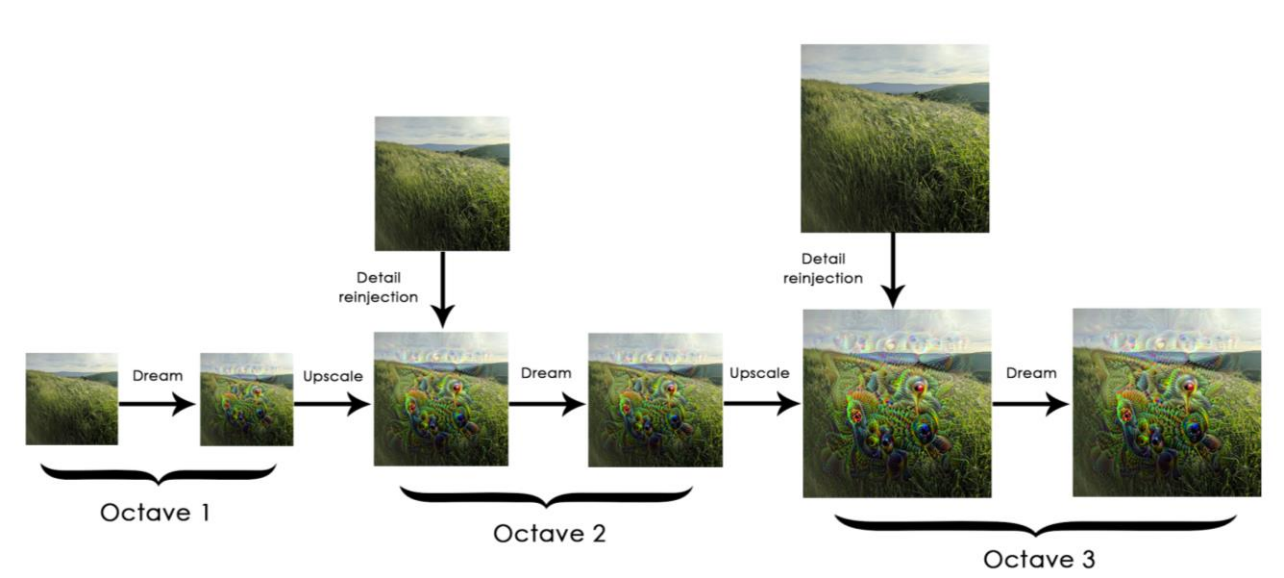


• The original image is processed at different scales "octaves" (typically a small number, e.g., 3, ~40% rescale size).

• For each octave we execute a "dream" loop, where layer activations (usually we take a set of neurons/layers) are computed for the input image. These normalized layer activations represent the "gradient" – we then execute a step of gradient ascent.

• We repeat several iterations of gradient ascent for the current octave, generating a scaled "dream" image. This image is the result of "over processing" the activations corresponding with the chosen layers/neurons for the current input.

Visualization: Dream Deeper



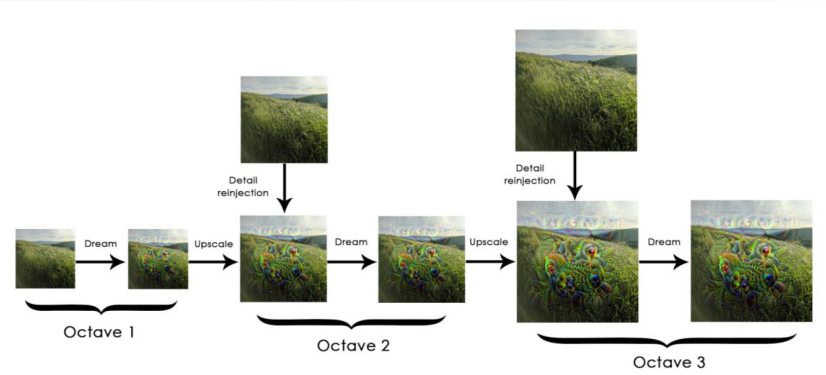
• After generating the "dream" image for the current octave, many of the fine-grain details of original image may be lost. For this reason, we perform a "detail injection", where the pixel difference between the upscaled image and original image is added to the dream image.

• We repeat this process for several octaves to generate the final DeepDream output.

Visualization: Dream Deeper

- DeepDream Loss Function

```python
def compute_loss(input_image):
    features = feature_extractor(input_image)
    # Initialize the loss
    loss = tf.zeros(shape=())
    for name in features.keys():
        coeff = layer_settings[name]
        activation = features[name]
        # We avoid border artifacts by only involving non-border pixels in the loss.
        scaling = tf.reduce_prod(tf.cast(tf.shape(activation), "float32"))
        loss += coeff * tf.reduce_sum(tf.square(activation[:, 2:-2, 2:-2, :])) / scaling
    return loss
```



- Gradient Ascent

```python
def gradient_ascent_step(img, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(img)
        loss = compute_loss(img)
    # Compute gradients.
    grads = tape.gradient(loss, img)
    # Normalize gradients.
    grads /= tf.maximum(tf.reduce_mean(tf.abs(grads)), 1e-6)
    img += learning_rate * grads
    return loss, img


def gradient_ascent_loop(img, iterations, learning_rate, max_loss=None):
    for i in range(iterations):
        loss, img = gradient_ascent_step(img, learning_rate)
        if max_loss is not None and loss > max_loss:
            break
        print("... Loss value at step %d: %.2f" % (i, loss))
    return img
```

- Looping Over Octaves

```python
original_img = preprocess_image(base_image_path)
original_shape = original_img.shape[1:3]

successive_shapes = [original_shape]
for i in range(1, num_octave):
    shape = tuple([int(dim / (octave_scale ** i)) for dim in original_shape])
    successive_shapes.append(shape)
successive_shapes = successive_shapes[::-1]
shrunk_original_img = tf.image.resize(original_img, successive_shapes[0])

img = tf.identity(original_img)  # Make a copy
for i, shape in enumerate(successive_shapes):
    print("Processing octave %d with shape %s" % (i, shape))
    img = tf.image.resize(img, shape)
    img = gradient_ascent_loop(
        img, iterations=iterations, learning_rate=step, max_loss=max_loss
    )
    upscaled_shrunk_original_img = tf.image.resize(shrunk_original_img, shape)
    same_size_original = tf.image.resize(original_img, shape)
    lost_detail = same_size_original - upscaled_shrunk_original_img

    img += lost_detail
    shrunk_original_img = tf.image.resize(original_img, shape)
```

https://github.com/google/deepdream

Visualization: Dream Deeper

Visual Attention: Captioning

• Human beings rarely use all the available sensory inputs in order to accomplish specific tasks.

• In particular, the retina contains a central **fovea** which has an extremely high resolution compared with the remainder of the eye. This region has a high concentration of color-sensitive cones, whereas most of the non-central portions of the eye have relatively low resolution with a predominance of color-insensitive rods.

• When, for instance, reading a street number, the fovea fixates on the number. Although one is aware of the other objects outside this central field of vision, it is virtually impossible to use images in the peripheral region to perform detail-oriented tasks.

Attention and saliency used for sign-reading in *Streetview* data (Google)

Visual Attention: Captioning

• The use of **visual attention** in computer vision has many viable applications, including segmentation, improving classification models, video tracking, text extraction, and medical imaging. Next, we focus on a common use case of attention: **image captioning**.

Show, Attend and Tell: Neural Image Caption Generation with Visual Attention

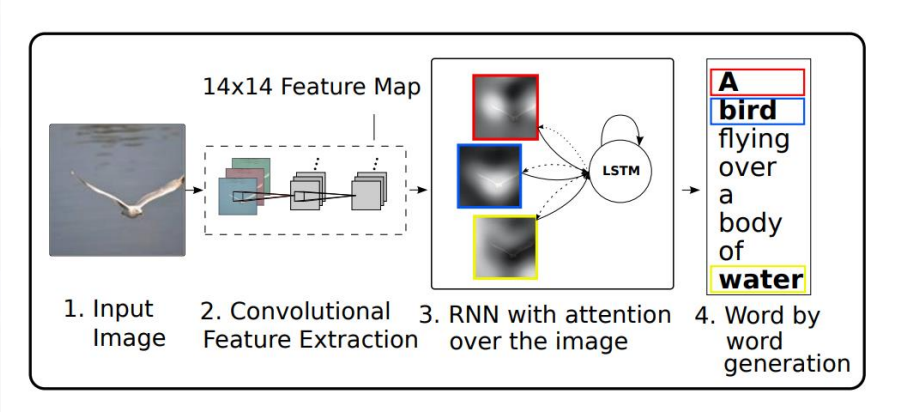
Kelvin Xu	KELVIN.XU@UMONTREAL.CA
Jimmy Lei Ba	JIMMY@PSI.UTORONTO.CA
Ryan Kiros	RKIROS@CS.TORONTO.EDU
Kyunghyun Cho	KYUNGHYUN.CHO@UMONTREAL.CA
Aaron Courville	AARON.COURVILLE@UMONTREAL.CA
Ruslan Salakhutdinov	RSALAKHU@CS.TORONTO.EDU
Richard S. Zemel	ZEMEL@CS.TORONTO.EDU
Yoshua Bengio	FIND-ME@THE.WEB

• Following the aforementioned "**coherence theory**" of vision (Rensink, 2000), the authors leverage visual attention so that the model learns to fix its "gaze" on salient objects while generating the corresponding words in the output sequence.

https://arxiv.org/pdf/1502.03044.pdf

Visual Attention: Captioning

• The authors introduce a multi-step algorithm: (1) CNN-based features are rendered for the input image, then (2) these features are fed into a Recurrent Neural Network (RNN) with attention mechanism that generates a caption.



(1) Concretely, given an input image, we extract L vector representations of different spatial regions using a pre-trained CNN; each vector is of dimension D:

$$a = \{a_1, ..., a_L\}, a_i \in \mathbb{R}^D$$

• Notably, the authors utilize **low-level features** to render this latent representation.

Visual Attention: Captioning

$$a = \{a_1, ..., a_L\}, a_i \in \mathbb{R}^D$$

• The goal is to produce a caption, i.e., a sequence of one-hot encoded vectors connoting words (from a vocabulary of K total words) in the caption:

$$y = \{y_1, ..., y_C\}, y_i \in \mathbb{R}^K$$

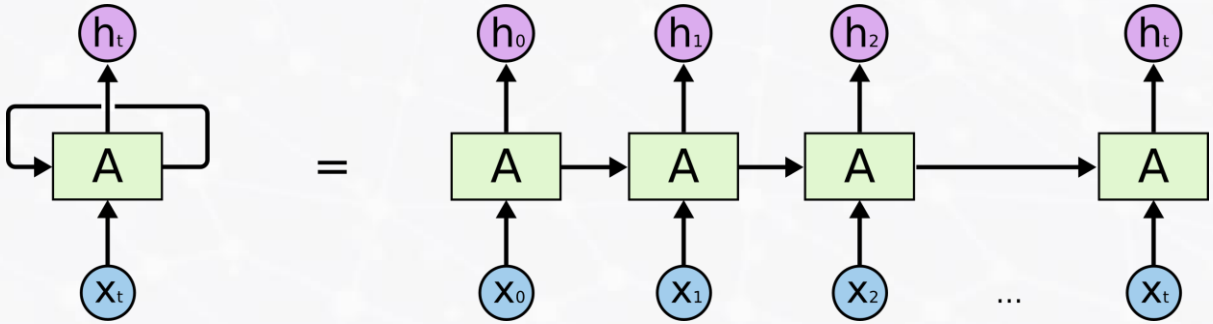
Visual Attention: Captioning

$$a = \{a_1, ..., a_L\}, a_i \in \mathbb{R}^D$$

• The goal is to produce a caption, i.e., a sequence of one-hot encoded vectors connoting words (from a vocabulary of K total words) in the caption:

$$y = \{y_1, ..., y_C\}, y_i \in \mathbb{R}^K$$

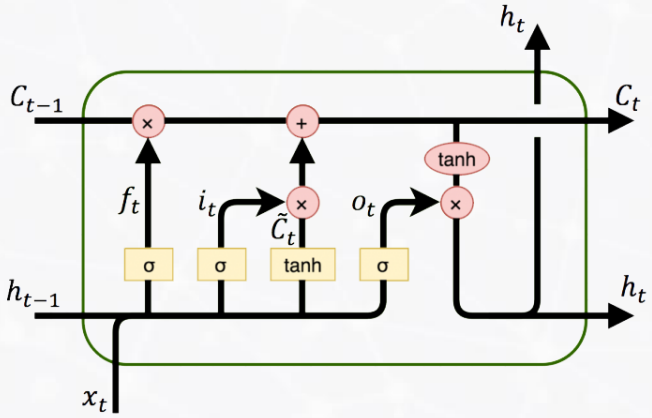
(2) The author employ an RNN to handle the caption generation/processing. RNNs represent traditional architectures for handling variable-size input/output; they are particularly well-suited for NLP problems for this reason.

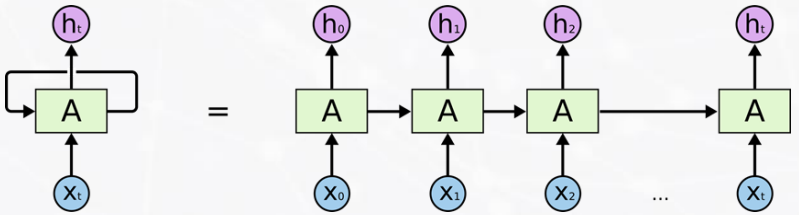


Visual Attention: Captioning

• In the late 1990s a variant of RNNs was introduced, called **Long-short term Memory** (LSTMs). These models greatly improved the capacity of RNNs for a variety of NLP-related tasks, including machine translation.

• In summary, an LSTM introduces several sub-network components, including **memory cells**, **forget gates**, and **input gates** (a multitude of additional nuanced LSTM model types exist), etc. The introduction of these components allows the RNN to learn "long-term" semantic dependencies between words (e.g., if, say, a dependent object in a sentence appears at the end of the sentence).

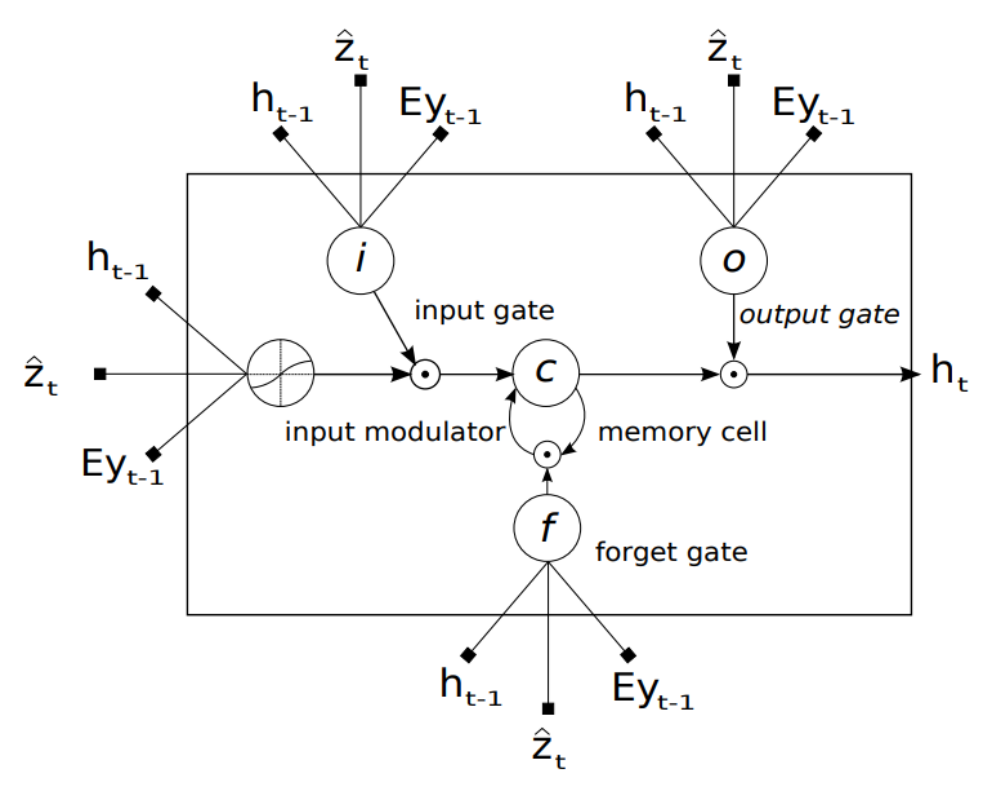




https://www.bioinf.jku.at/publications/older/2604.pdf

Visual Attention: Captioning

• In the LSTM cell, h_t is the **hidden representation** of the LSTM (at step t); $y_t \in \mathbb{R}^k$ is the **one-hot encoded word** in the caption (at step t); $E \in \mathbb{R}^{m \times k}$ is a learned embedding matrix to map y_t to an embedding space (Ey_t); $\hat{z}_t \in \mathbb{R}^L$ is the **context vector** (explained next), capturing the visual information associated with a particular input location.

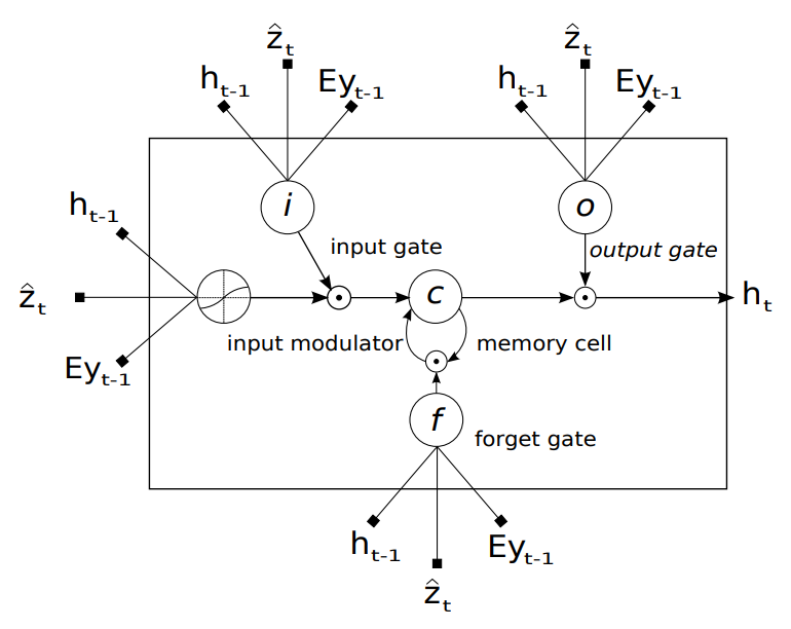


https://arxiv.org/pdf/1502.03044.pdf

Visual Attention: Captioning

• In the LSTM cell, h_t is the **hidden representation** of the LSTM (at step t); $y_t \in \mathbb{R}^k$ is the **one-hot encoded word** in the caption (at step t); $E \in \mathbb{R}^{m \times k}$ is a learned embedding matrix to map y_t to an embedding space ($\mathrm{E}y_t$); $\hat{z}_t \in \mathbb{R}^L$ is the **context vector** (explained below), capturing the visual information associated with a particular input location.

• The input to the LSTM cell consists of the context vector ($\hat{z}_t$), word embedding ($\mathrm{E}y_{t-1}$), and previous hidden state (h_{t-1}). A combination of learnable affine transformations and a non-linear sigmoid transformations modulate the input, output, and forget gates within the LSTM.

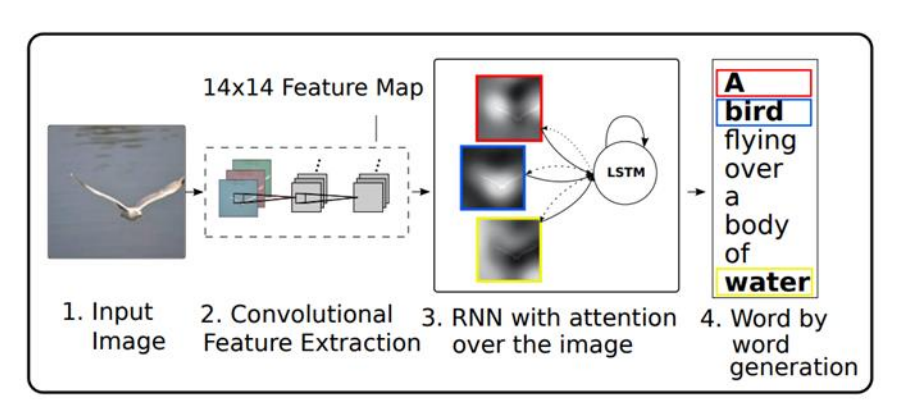


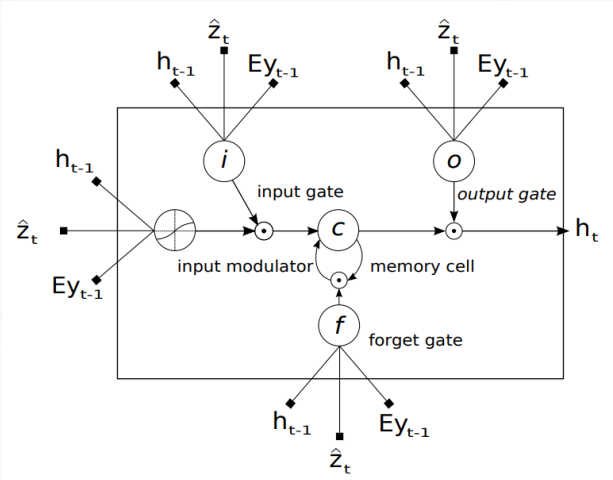
https://arxiv.org/pdf/1502.03044.pdf

Visual Attention: Captioning

• Using the hidden representation output from the LSTM (h_{t-1}) in conjunction with annotation vectors (a_i) derived from the low-level features of a CNN (for patch i), the authors train a separate (MLP) attention model: $f_{att}(a_i, h_{t-1})$ that returns a weight associated with each hidden representation and each annotation vector (associated with different image patches in the input image).

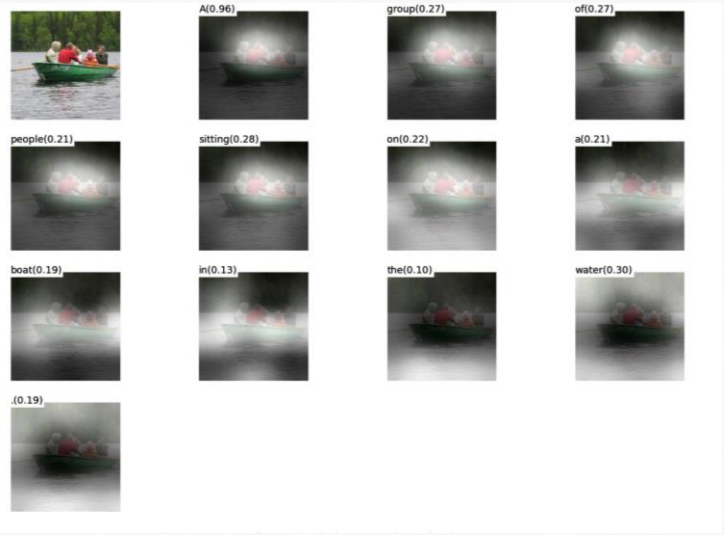
• The output of $f_{att}(\cdot)$ is a context vector corresponding with time t: $\hat{z}_t \in \mathbb{R}^L$, which weights the **visual attention** corresponding with the relevant portion of the image for the current caption word. Depending on the formulation of $f_{att}(\cdot)$, the visual attention context vector can be "hard" (in which case a single region of attention is highlighted) or "soft" (in which case multiple regions of attention are highlighted).





https://arxiv.org/pdf/1502.03044.pdf

Visual Attention: Captioning



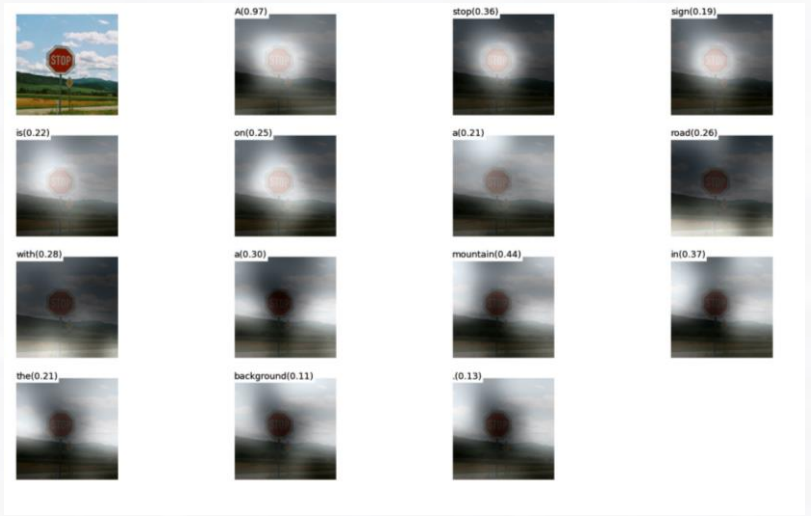
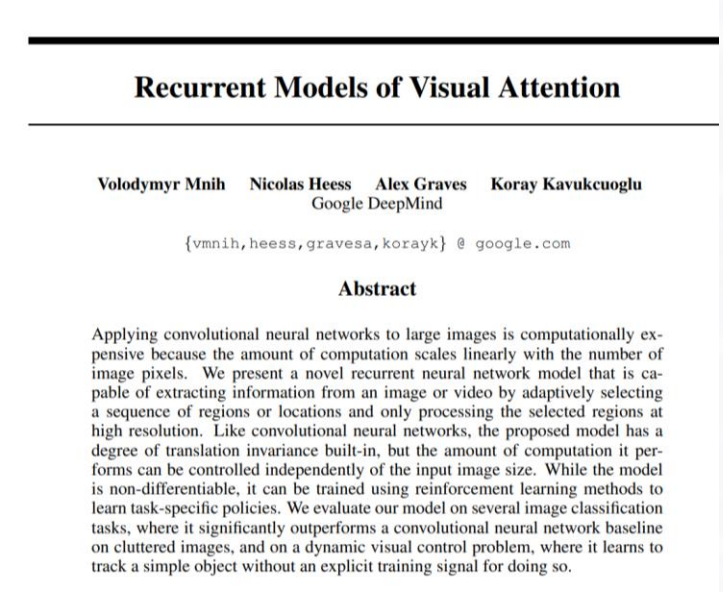


Figure 5. Examples of mistakes where we can use attention to gain intuition into what the model saw.

https://arxiv.org/pdf/1502.03044.pdf

Visual Attention: Classification

• A closely-related work leveraging attention for image classification is the so-called *Glimpse Network* (DeepMind, 2014).



Recurrent Models of Visual Attention

Volodymyr Mnih **Nicolas Heess** **Alex Graves** **Koray Kavukcuoglu**
Google DeepMind

{vmnih,heess,gravesa,korayk} @ google.com

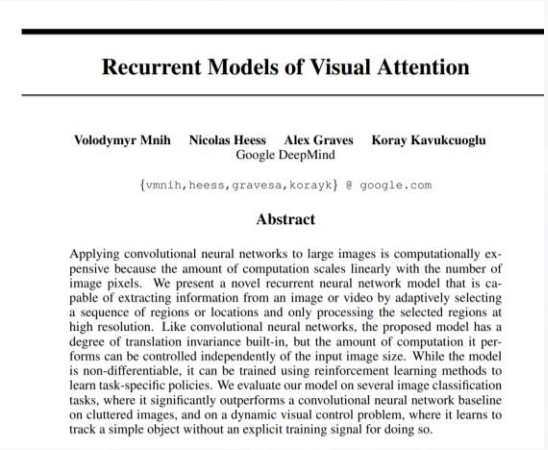
Abstract

Applying convolutional neural networks to large images is computationally expensive because the amount of computation scales linearly with the number of image pixels. We present a novel recurrent neural network model that is capable of extracting information from an image or video by adaptively selecting a sequence of regions or locations and only processing the selected regions at high resolution. Like convolutional neural networks, the proposed model has a degree of translation invariance built-in, but the amount of computation it performs can be controlled independently of the input image size. While the model is non-differentiable, it can be trained using reinforcement learning methods to learn task-specific policies. We evaluate our model on several image classification tasks, where it significantly outperforms a convolutional neural network baseline on cluttered images, and on a dynamic visual control problem, where it learns to track a simple object without an explicit training signal for doing so.

https://proceedings.neurips.cc/paper/2014/file/09c6c3783b4a70054da74f2538ed47c6-Paper.pdf

Visual Attention: Classification

• A closely-related work leveraging attention for image classification is the so-called *Glimpse Network* (DeepMind, 2014).



Recurrent Models of Visual Attention

Volodymyr Mnih **Nicolas Heess** **Alex Graves** **Koray Kavukcuoglu**
Google DeepMind

{vmnih,heess,gravesa,korayk} @ google.com

Abstract

Applying convolutional neural networks to large images is computationally expensive because the amount of computation scales linearly with the number of image pixels. We present a novel recurrent neural network model that is capable of extracting information from an image or video by adaptively selecting a sequence of regions or locations and only processing the selected regions at high resolution. Like convolutional neural networks, the proposed model has a degree of translation invariance built-in, but the amount of computation it performs can be controlled independently of the input image size. While the model is non-differentiable, it can be trained using reinforcement learning methods to learn task-specific policies. We evaluate our model on several image classification tasks, where it significantly outperforms a convolutional neural network baseline on cluttered images, and on a dynamic visual control problem, where it learns to track a simple object without an explicit training signal for doing so.

• Glimpse uses reinforcement learning (RL) to focus visual attention on relevant parts of an image. The general idea, borrowed loosely from *coherence theory,* is that computer vision tasks (classification in this case) can be made more efficient through the incorporation of an attention mechanism.

• In this case, the model extracts image information by adaptively selecting sub-regions (or frames in a video) for processing – in lieu of processing an entire image/video at once.

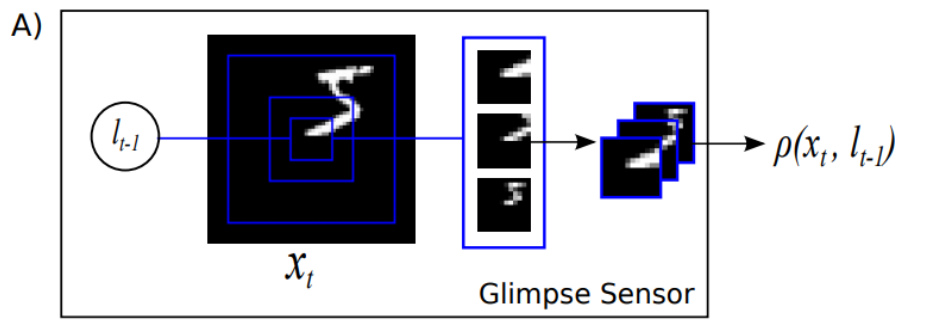
https://proceedings.neurips.cc/paper/2014/file/09c6c3783b4a70054da74f2538ed47c6-Paper.pdf

Visual Attention: Classification

• The authors use an RNN as the controller to identify the precise location of visual attention in each time-stamp; this choice is based on the feedback from the glimpse in the previous time-stamp.

• This work show that using the glimpse network equipped with a visual attention mechanism can outperform a generic CNN for classification.

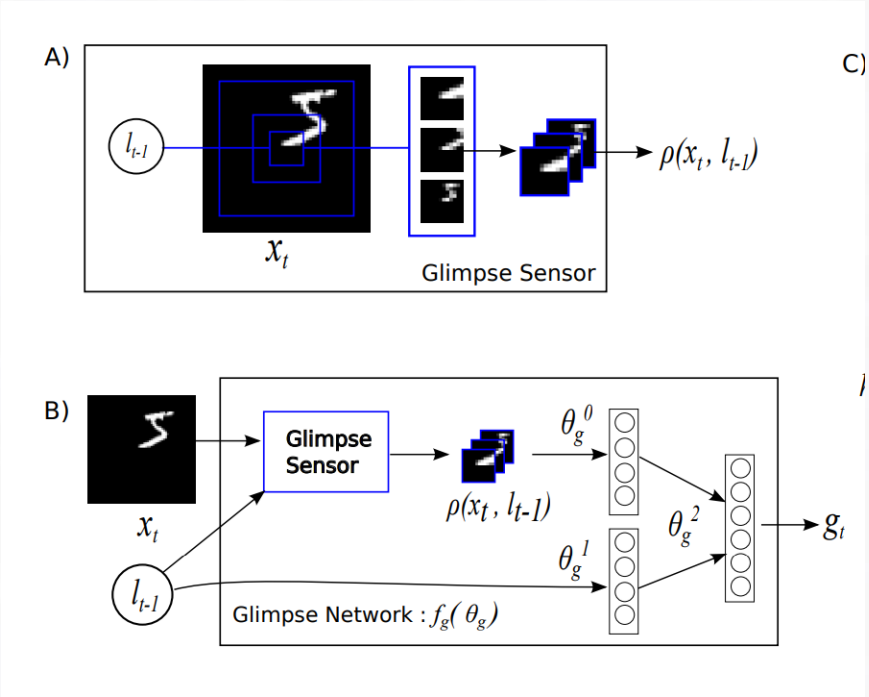
Visual Attention: Classification



(A) **Glimpse sensor**: Given an image with representation X_t, a glimpse sensor creates a retina-like representation of the image. The glimpse sensor is conceptually assumed to not have full access to the image (because of bandwidth constraints) and is thus able to access only a small portion of the image in high-resolution, centered at l_{t-1}.

• The resolution of a particular location in the image reduces with the distance from the location l_{t-1}. This reduced representation is denoted $\rho(X_t, l_{t-1})$.

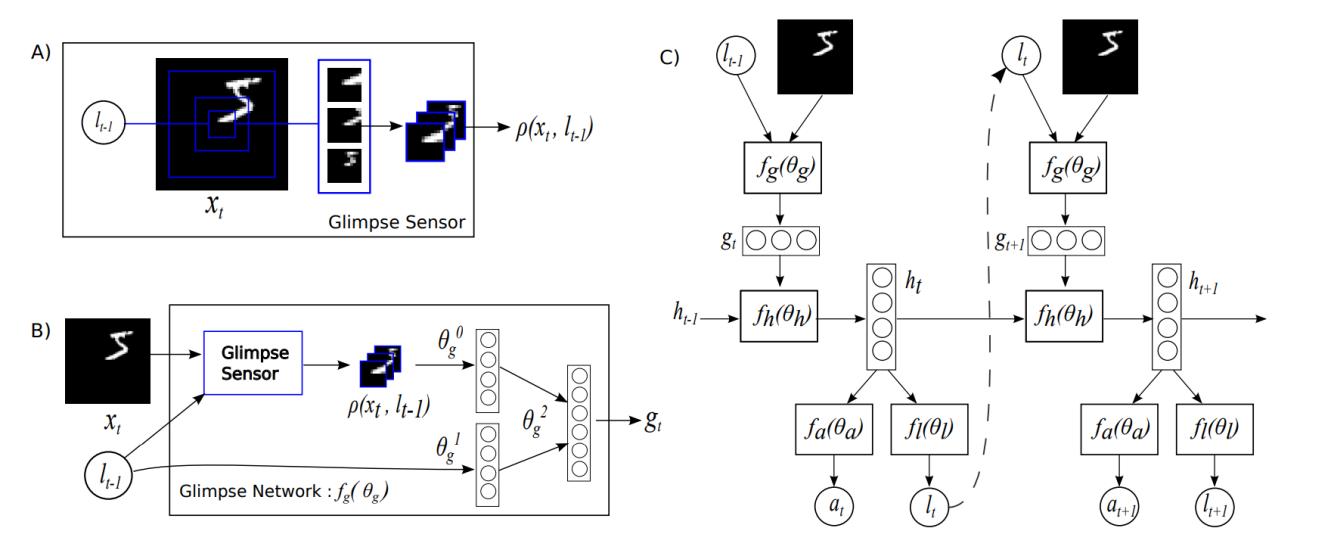
Visual Attention: Classification



(B) **Glimpse network**: The glimpse network contains the glimpse sensor and encodes both the glimpse location l_{t-1} and the glimpse representation ρ into latent space using simple linear layers.

• Subsequently, the two are combined into a single hidden representation using another linear layer. The resulting output g_t is the input to the tth time-stamp of the hidden layer in the RNN.

Visual Attention: Classification



(C) **RNN**: the RNN includes the glimpse network and sensor; it generates action-location output pairs at each time-step. The **action output** at time-stamp t is denoted by a_t, and rewards are associated with the action (to train the RL model); a reward might be associated, say, with the class label of the object or numerical digits (for Streetview data).

• The RNN also outputs l_t the location in the image for the next time-stamp on which the glimpse network should focus. Training of the RNN is done to **maximize the expected reward** over time (e.g., predict the correct class). Tunable parameters include: θ_g (glimpse network parameters), θ_h (RNN hidden network parameters) and θ_a (action network parameters).

Fin