CS 231n Summary

1. KNN

motivation: to build an image classifier, we intend to minimize the "distance" of two images so that they are classified as the same category.

approach: store the entire training set into the memory and predict which label a test image should be classified by comparing the L1 or L2 distance and find the top k closest images. Each image votes on the label of the test image and we choose the label with the highest score. e.g. suppose k = 7, the top 7 images are categorized as label 3, 2, 1, 2, 2, 1, 2, then the test image should be labeled as class 2.

metrics:

L1 distance:

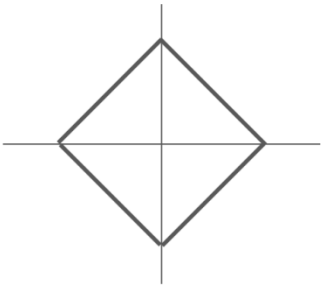
$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

L2 distance:

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

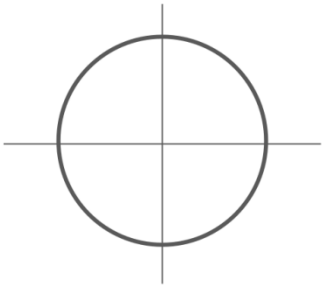
L1 (Manhattan) distance

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



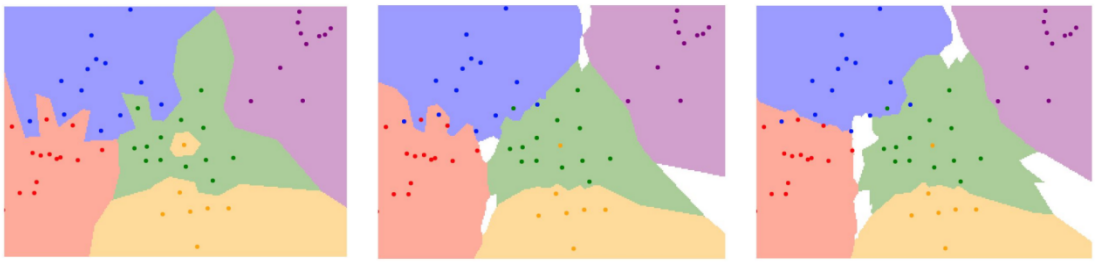
L2 (Euclidean) distance

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$



why choose top k images instead of the closest image?

Because we want our decision boundary looks more natural and smoother even some outliers in the dataset.

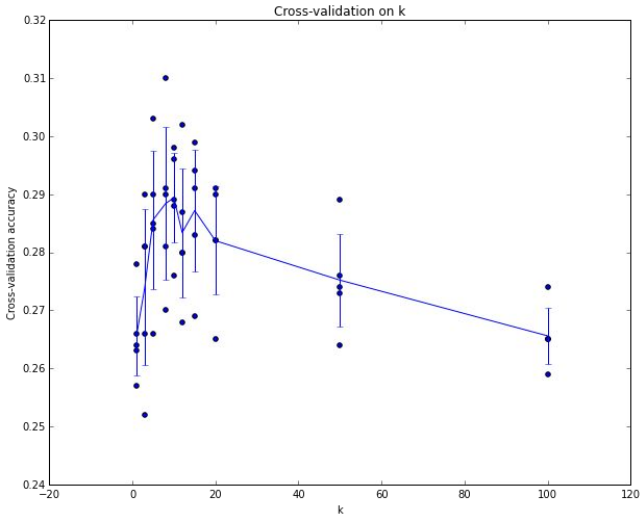


pros & cons:

pros: easy to implement and understand, no need to train

cons: computational cost at test time, must choose a best k value(during cross validation), low accuracy(same

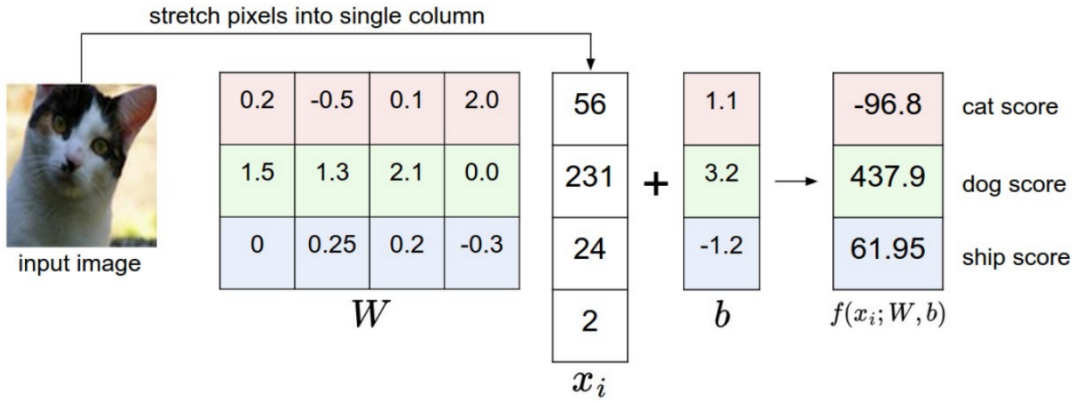
background may be classified as the same label)



2. Linear Classifier

motivation: a linear function mapping from an image to a class label. Specifically, find a function f(x) parameterized by weight W and bias b, thus we can use the formula to predict $y = Wx + b$.

score value: if it is an image classification problem, the first thing we should do is flatten the 3-D tensor of an image into a 1-D column vector: (width, height, channel) → (width*height*channel), then compute the score value for each class.



weight matrix W is interpreted as a template for each class. When a new image is tested, the image is to choose a row of weight matrix that fits it best.

Loss function: to evaluate the weight matrix W(i.e. how bad of the model), we need a function to tell us the scale of the deviation between predicted score value and ground truth score label. So the two loss functions(SVM, Softmax) are the most popular ones.

(i) SVM:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

where s_j is the score value of the j-th example, s_{y_i} is the score value of the correct label, Δ is the fixed margin which indicates the loss is zero only if the difference between s_j and s_{y_i} is higher than the margin Δ.

average loss:

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, W_j^T x_i - W_{y_i}^T x_i + \Delta)$$

derivatives:

$$\nabla_{W_j} L(W) = \frac{1}{N}\sum_{i=1}^{N} x_i^T \quad \text{if } W_j^T x_i - W_{y_i}^T x_i + \Delta) > 0$$

$$\nabla_{W_{y_i}} L(W) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i} -x_i^T \quad \text{if } W_j^T x_i - W_{y_i}^T x_i + \Delta) > 0$$

(ii) Softmax:

$$L_i = -\log(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}})$$

where s_j is the score value of the j-th unit, s_{y_i} is the score value of the correct label, the term $\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}$ represents the normalized probabilities of each class. The negative log likelihood measures the scale of difference between a specific label and ground truth label, so that if the value is closer to 1, the loss is smaller. Otherwise, the loss grows exponentially.

average loss:

$$L = \frac{1}{N}\sum_{i=1}^{N} -\log(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}})$$

derivatives:

$$\nabla_{W_j} L(W) = \frac{1}{N}\sum_{i=1}^{N} \frac{e^{s_j}}{\sum_j e^{s_j}} x_i^T$$

$$\nabla_{W_{y_i}} L(W) = \frac{1}{N}\sum_{i=1}^{N} \left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}} - 1\right) x_i^T$$

regularization: to prevent overfitting, a regularization term $R(W)$ is added to loss function.

Typically, we use L2 regularization, thus the loss function can be denoted to:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \frac{\lambda}{2}\sum_k\sum_l W_{k,l}^2$$

where the first term is the data loss shown above, and the second term is regularization term. Regularization strength is weighted by λ, a hyperparameter determined by cross-validation.

derivative:

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i + \lambda\sum_k\sum_l W_{k,l}$$

optimization: The loss function quantifies the quality of weight matrix W, but our goal is to minimize our loss function by changing the value in W. There are two strategies helping us refine the weight matrix W: one is Random Search another is Gradient Descent.

Random Search: a very naive method, simply randomly initialize weight matrix W to improve the total accuracy.

Gradient Descent: The idea is to compute partial derivatives with respect to each entry of weight matrix W, and

update them by following the descent of the gradient.

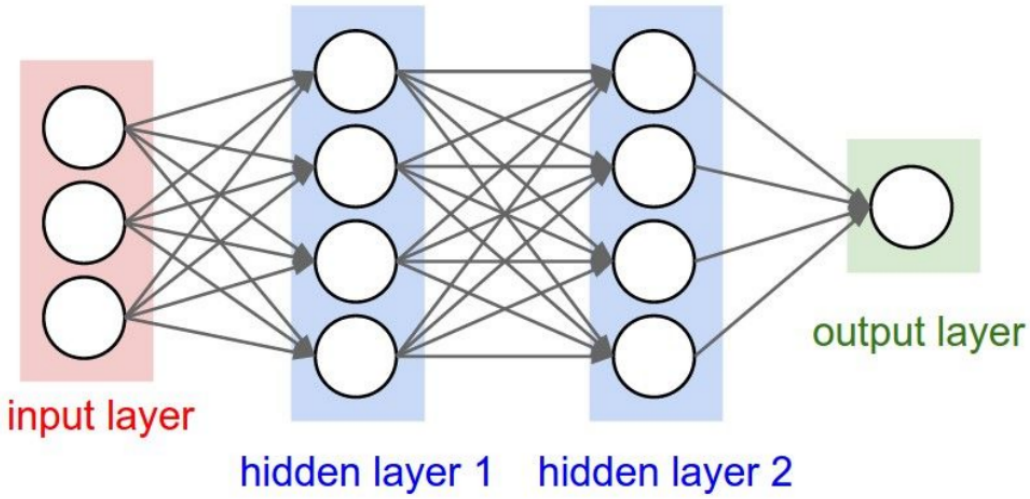
$$W = W - lr * \nabla_W L(W)$$

where lr is a hyperparameter which means step size(learning rate) for each update.

3. Neural Network:

motivation: instead of a single linear classifier, a neural network with several layers can represent even more complex features and help us fit sophisticated features.

architecture: an input layer that accept the dataset, several hidden layer to represent complex features and an output layer which output the result.



forward pass: the process of the input data fed into the neural network to output the predicted value. Each process the data feed-forward contains two steps: linear transformation(matrix multiplication) and activation(non-linearity transformation).

Suppose the neural network has one hidden layer, the input is x, the weight matrix and bias vector for each transformation is W_1, W_2, b_1, b_2 respectively, so a series of functions can be written as follows.

$$f_h = W_1 x + b_1$$

hidden layer, after activation(ReLU):

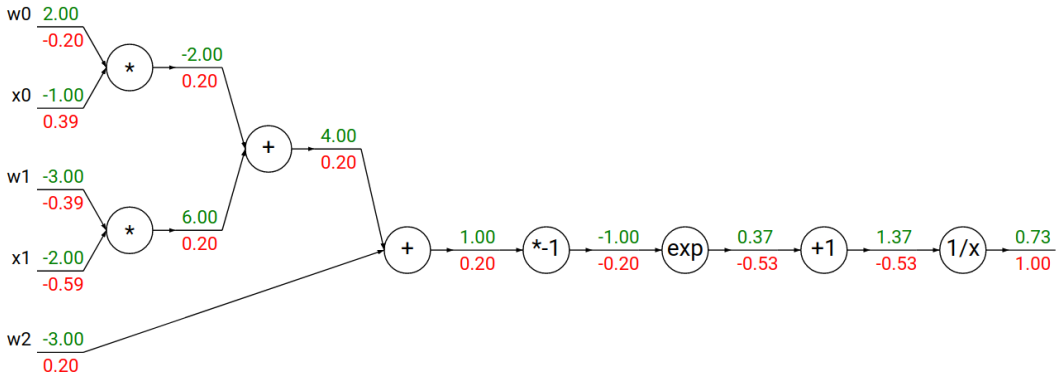
$$f_h = \max(W_1 x + b_1, 0)$$

output layer:

$$f_o = W_2 \max(W_1 x + b_1, 0) + b_2$$

backward pass: to minimize the objective function(loss function), we need to update each layer's weight matrix $W^{[l]}$ and bias vector $b^{[l]}$. To this end, the partial derivatives with respect to each layer's parameter should be computed based on the law of calculus(chain rule).

e.g. find the derivatives of the function $f(x; w) = \frac{1}{1+e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$ with respect to each parameter.



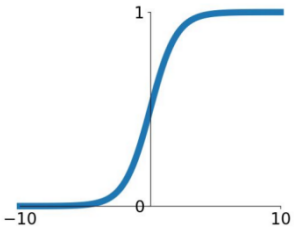
stochastic gradient descent(mini-batch gradient descent):

Instead of inputting the whole dataset into the neural network and updating only once for each epoch, we randomly sample a batch of data in the dataset, find the gradient and update parameters within the batch.

activation function: sometimes linear transformation cannot fit in all cases, especially those non-linear function. e.g. it is difficult to learn a curve, say, $y = x^2$, if we only use linear function to fit it. Therefore, such non-linearities as Sigmoid, Tanh, ReLU are helpful to activate some specific neurons for better mapping to our target.

(i) Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



Sigmoid

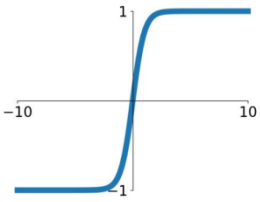
derivative:

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}}\right)\left(\frac{1}{1 + e^{-x}}\right) = (1 - \sigma(x))\sigma(x)$$

drawbacks: ① outputs are not zero-centered ② gradient saturation when output is too large or small ③ computational cost for $e^{(\cdots)}$.

(ii) Tanh

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



tanh(x)

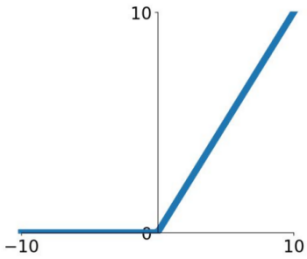
derivative:

$$\frac{df(x)}{dx} = \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} = \frac{4}{(e^x + e^{-x})^2} = 1 - \tanh^2(x)$$

how: ① zero-centered ② gradient saturation for a very large and small output ③ computational cost for $e^{(\cdots)}$.

(iii) ReLU

$$f(x) = \max(0, x)$$



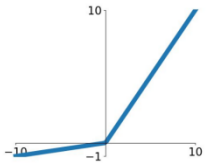
ReLU

derivative:

$$\frac{df(x)}{dx} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}$$

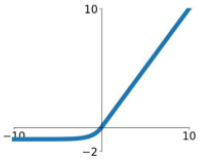
how: ① computationally efficient ② do not saturate ③ neuron will die if the output is negative

to solve dead neurons, we can use other variant, such as leaky ReLU, Exponential Linear Units.



Leaky ReLU

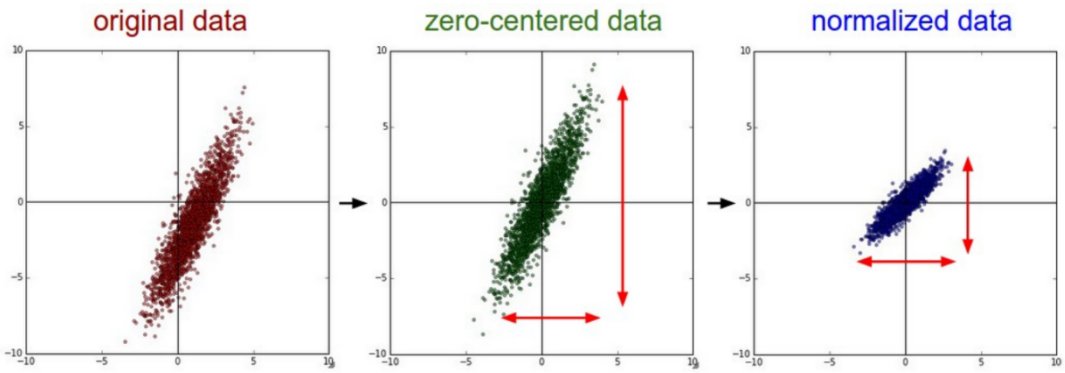
$$f(x) = \max(0.01x, x)$$



Exponential Linear Units

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$$

data preprocessing: to achieve a better performance, sometimes the data must be zero-centered, and standard deviation is equal to 1 for each dimension. For image, in particular, it is better to let the pixel value be in the range of (0, 1)(suppose each color channel is represented by 8-bit, 256 possible values).

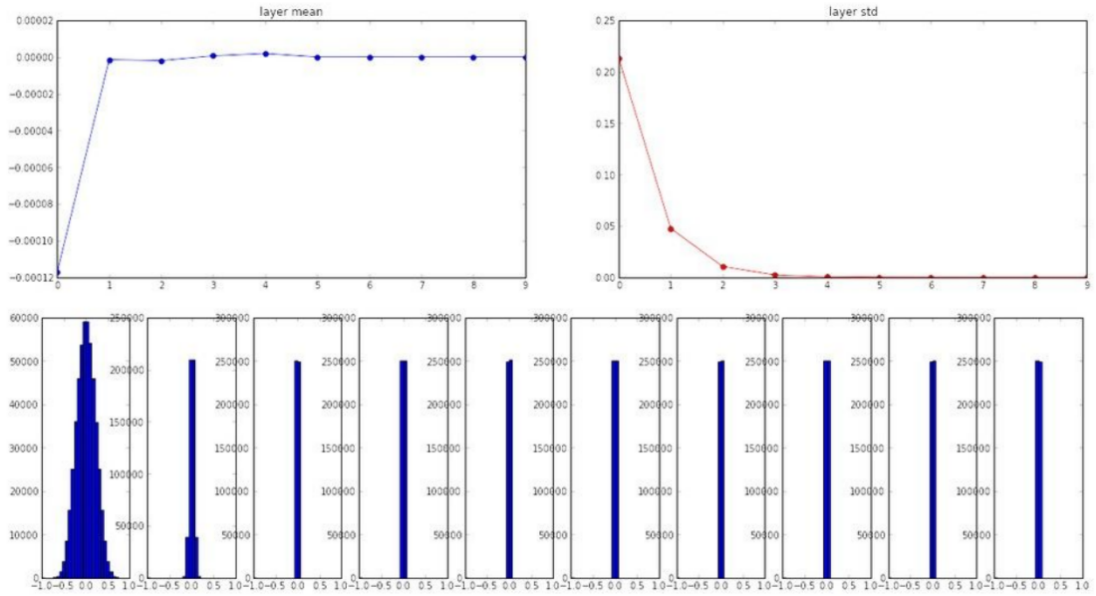


weight initialization: at the start of training, the weight matrix must be initialized to a certain number. A very bad idea is initializing weights to 0, because when backpropagating error gradient, each neuron would obtain the same

gradient, so that they undergo the exact same parameter updates.

naive idea: small random numbers; works well for shallow neural network, but for deeper network, all activations become zero, so the parameter updates for each layers weight become zero.

Xavier initialization: w = np.random.randn(n) / sqrt(n), where n is the number of input.



why: the variance is proportional to number of input, so we want the variance of output for each neuron to 1. To do that, the weight must be initialized along with the calibration term. The derivation is as follows:

$$\mathrm{Var}(s) = \mathrm{Var}(\sum_i^n w_i x_i)$$

$$= \sum_i^n \mathrm{Var}(w_i x_i)$$

$$= \sum_i^n [E(w_i)]^2 \mathrm{Var}(x_i) + E[(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i)$$

$$= \sum_i^n \mathrm{Var}(x_i)\mathrm{Var}(w_i)$$

$$= (n\mathrm{Var}(w))\,\mathrm{Var}(x)$$

assume the variance of w and x are 1 and identically distributed, if $\mathrm{Var(s)} = (\mathrm{nVar(w)Var(x)})$, then $\mathrm{Var(w)}$ should be $1/\mathrm{n}$, so it implies that for each $\mathrm{w_i}$, it must multiply a scalar $1/\sqrt{\mathrm{n}}$.

ReLU initialization: w = np.random.randn(n) / sqrt(n/2)

batch normalization: our goal is the data is Gaussian distribution before flowing into non-linearity. To do that, we normalize them manually for each layer.

formula:

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

where γ and β are learnable parameter during backpropagation, which measure the scale of the normalization and the shift of mean value.

derivative:

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$

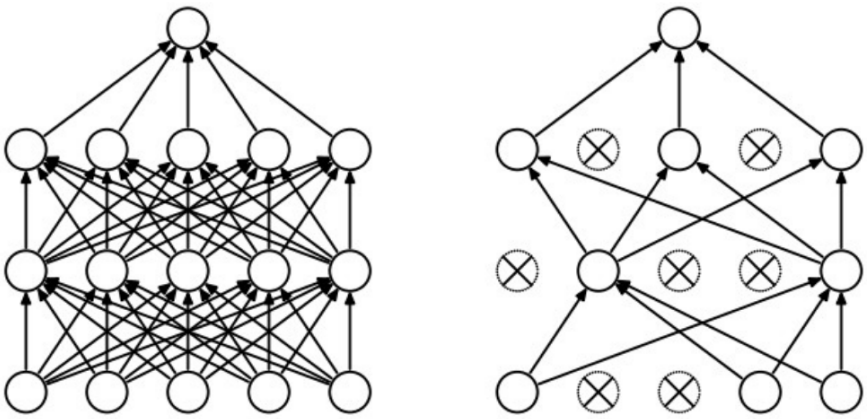
$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

dropout: a effective regularization technique implemented by keeping a neuron active with the probability of p and setting to zero with the probability of $(1-p)$.



(a) Standard Neural Net (b) After applying dropout.

At test time, we do not want to output a random value, so it needs to output the expected value during test time by multiplying the probability p.

$$f(x) = E_z[f(x,z)] = \int p(z) f(x,z) dz$$

parameter update: Once the gradient is computed with backpropagation, the gradients are used to perform a parameter update. Several approaches are listed below for updating parameters.

(i) SGD

a naive approach, simply update a step by following the gradient.

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

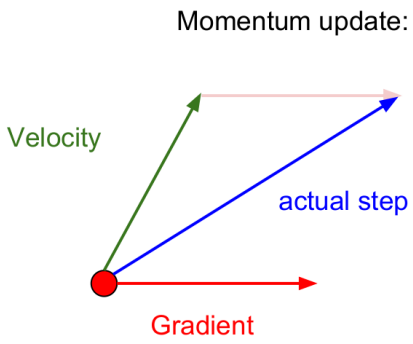
where α is the learning rate or step size.

(ii) SGD+Momentum

an analogy is a ball slides from the top of slope at a certain initial velocity, so the momentum is the current velocity, and the gradient is the force felt by the ball. In parameter update, we add the current velocity and gradient together as the updating direction.

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

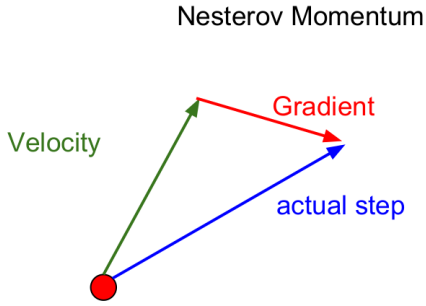
where ρ is a hyperparameter which is similar to friction.



(iii) Nesterov Momentum

an variant of SGD+Momentum, the slight difference is instead of computing the gradient at the current position, we "look ahead" at the approximate position $x_t + \rho v_t$ and compute the gradient at that position, then add them together.

$$v_{t+1} = \rho v_t + \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t - v_{t+1}$$



(iv) Adagrad

an adaptive gradient descent method aiming at addressing the issue that gradient does not point to the direction that minimizes gradient. In other words, some direction gains a very high gradient, while others have a lower one. To handle that, we normalize the parameter update step, element-wise.

$$n_{t+1} = n_t + \nabla f(x_t)^2$$
$$x_{t+1} = x_t - \frac{\alpha \nabla f(x_t)}{\sqrt{n_{t+1} + \epsilon}}$$

where ϵ (range from 10^{-4} to 10^{-8})is the smoothing term to avoid division by 0.

(v) RMSprop

a variant version of Adagrad. To avoid accumulative effect(aggressive learning rate decline) of square of gradient, a decay rate ρ is added for each update of cache.

$$n_{t+1} = \rho n_t + (1-\rho)\nabla f(x_t)^2$$

$$x_{t+1} = x_t - \frac{\alpha \nabla f(x_t)}{\sqrt{n_{t+1} + \epsilon}}$$

(vi) Adam

Adam can be regarded as RMSprop with momentum.

$$v_{t+1} = \beta_1 v_t + (1-\beta_1)\nabla f(x_t)$$

$$n_{t+1} = \beta_2 n_t + (1-\beta_2)\nabla f(x_t)^2$$

$$x_{t+1} = x_t - \alpha \frac{v_{t+1}}{\sqrt{n_{t+1} + \epsilon}}$$

where β_1 and β_2 are two hyperparameters that are recommended to set 0.9 and 0.999 respectively.

Normally, we initialize v_t and n_t to 0, and the problem is in the first few time step, the value v and n are prone to 0, so the term v_{t+1} and n_{t+1} needs to be normalized to obtain a higher gradient.

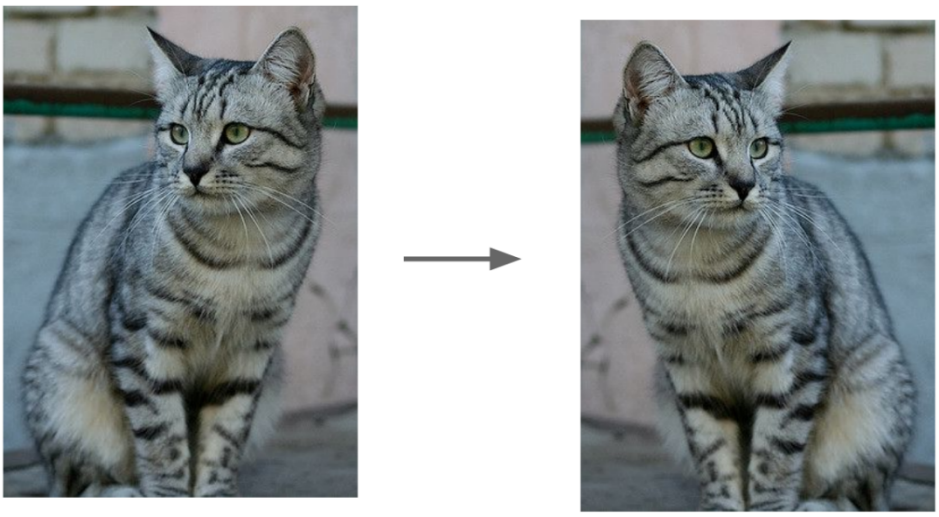
$$v'_{t+1} = \frac{v_{t+1}}{1-\beta_1^t}$$

$$n'_{t+1} = \frac{n_{t+1}}{1-\beta_2^t}$$

$$x_{t+1} = x_t - \alpha \frac{v'_{t+1}}{\sqrt{n'_{t+1} + \epsilon}}$$

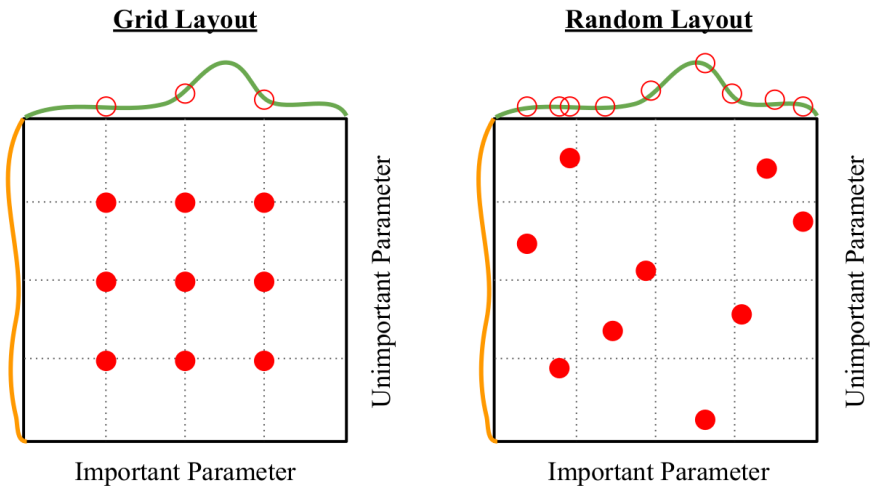
where t is the time step from 1 to ∞.

data augmentation: a method for regularization: cropping, shearing, rotation, flipping, distortion, color jitter, etc.



hyperparameter tuning tricks

Prefer random search to grid search, because random search allows you to explore better values for some important hyperparameters.



convolutional neural network: use 3D volumes of neurons to build a feature map to share parameters w and b, so that we can reduce computational cost.

(i) convolutional layer

operation: each filter slides over the image spatially, and compute the dot product.

$$n'_H = \left\lfloor \frac{n_H + 2 * \text{padding} - \text{kernel size}}{\text{stride}} \right\rfloor + 1$$

$$n'_W = \left\lfloor \frac{n_W + 2 * \text{padding} - \text{kernel size}}{\text{stride}} \right\rfloor + 1$$

n'_C = the number of kernels

(ii) pooling layer

to reduce the size of output volume, pooling layer is often used.

max-pooling: choose the maximum value within the volume

(iii) batch-norm layer

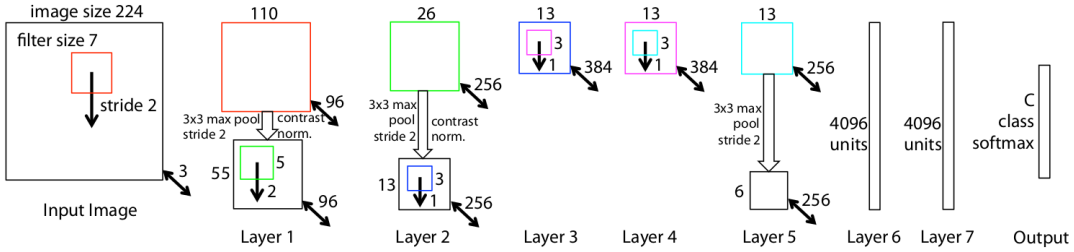
before flowing into non-linearity, the value of 3D volume for each mini-batch needs to be normalized.

CNN architecture: AlexNet, ZFNet, VGG, GoogLeNet, ResNet

① AlexNet:

[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

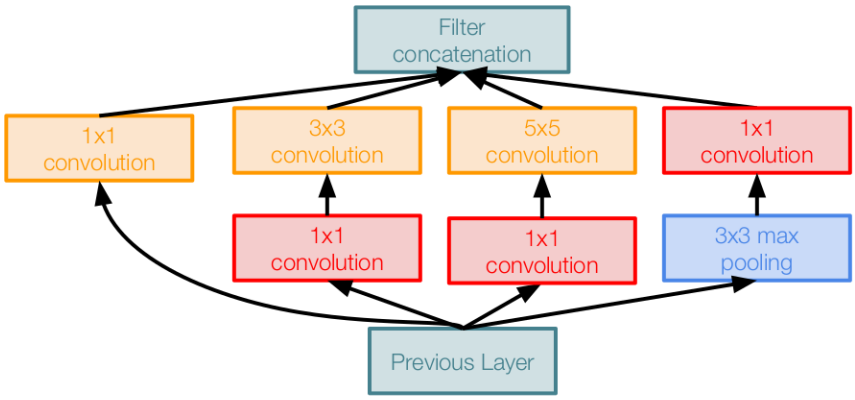
② ZFNet:



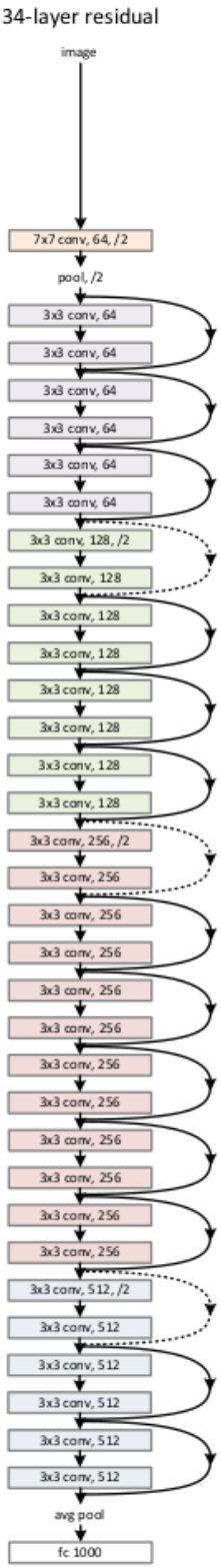
③ VGG

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

④ GoogLeNet

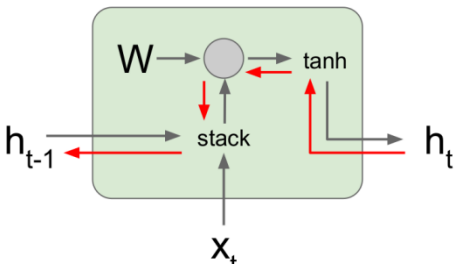


⑤ ResNet



recurrent neural network

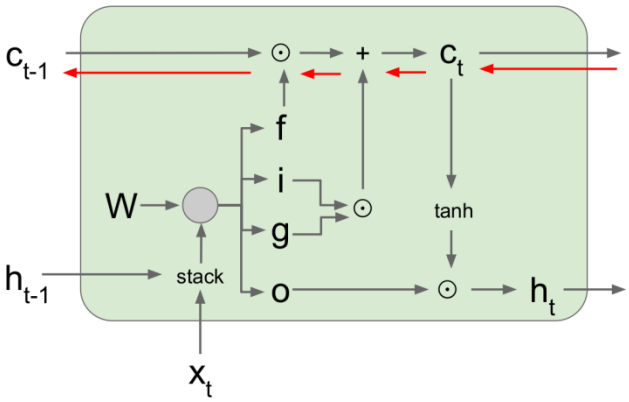
① Vanilla RNN



$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$= \tanh\left((W_{hh} \quad W_{hx})\begin{pmatrix}h_{t-1}\\x_t\end{pmatrix}\right)$$
$$= \tanh\left(W\begin{pmatrix}h_{t-1}\\x_t\end{pmatrix}\right)$$

issue: gradient may explode or vanish when backpropagation

② LSTM



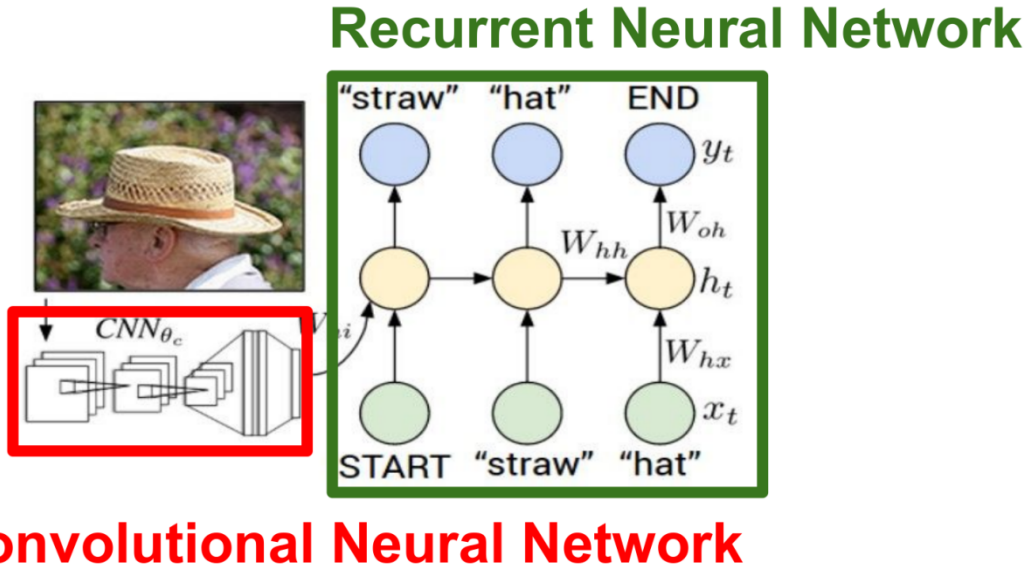
$$\begin{pmatrix}i\\f\\o\\g\end{pmatrix} = \begin{pmatrix}\sigma\\\sigma\\\sigma\\\tanh\end{pmatrix} W \begin{pmatrix}h_{t-1}\\x_t\end{pmatrix}$$
$$c_t = f \odot c_{t-1} + i \odot g$$
$$h_t = o \odot \tanh(c_t)$$

gradient can flow from c_t to c_{t-1} directly, thus alleviate gradient explosion and gradient vanish, which is similar to residual block in ResNet.

image captioning

extract feature vector from a state-of-the-art model(e.g. ResNet) as initial hidden state h_0 When training, each word of a caption of corresponding image is fed into RNN model(e.g. LSTM) at each time series and the output is also the caption but shift one word left.

When testing, the extracted feature vector from CNN is also as the hidden state h_0, but each word is generated by sampling. The most probable word in the previous time frame should be the input of the current time step.



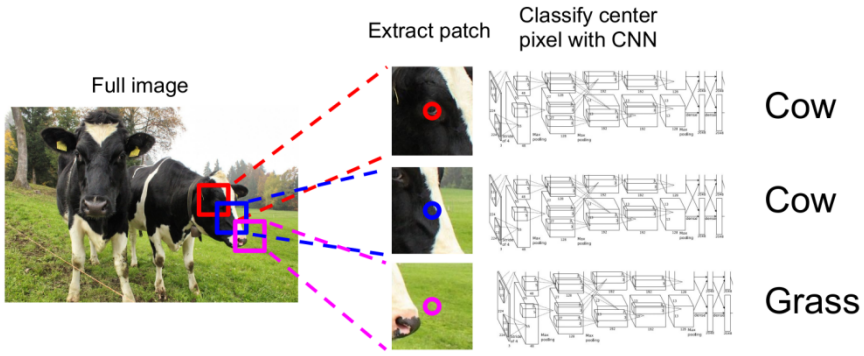
4. Detection and Segmentation

motivation: a single image classifier is not enough for real-world tasks. Not only the model is needed for labeling an specific image, but also detecting which portion has an object and drawing a bounding box to localize these objects.

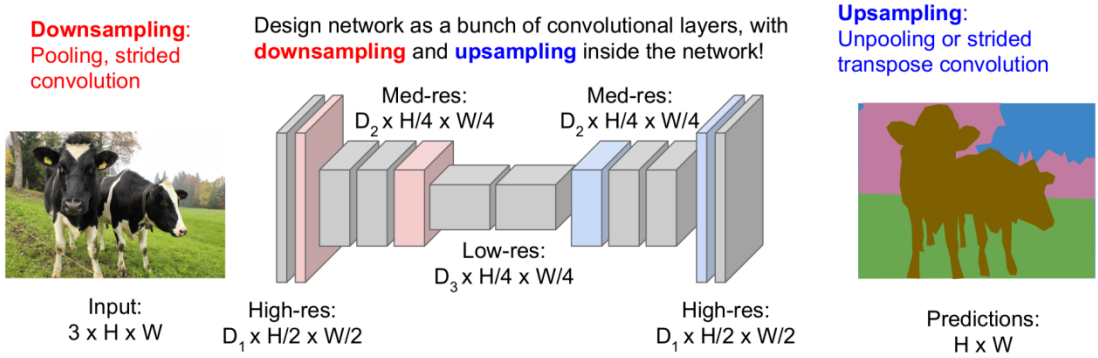
tasks: semantic segmentation, object detection, instance segmentation

semantic segmentation: our goal is label each pixel in a given image with category label.

ideas: ① sliding window: choose a certain rectangular portion as a window, slide across the image and feed each window into the neural network before making a prediction. Each pixel within the window will be labeled as corresponding category. Problem: inefficient, not reuse shared features between patches.



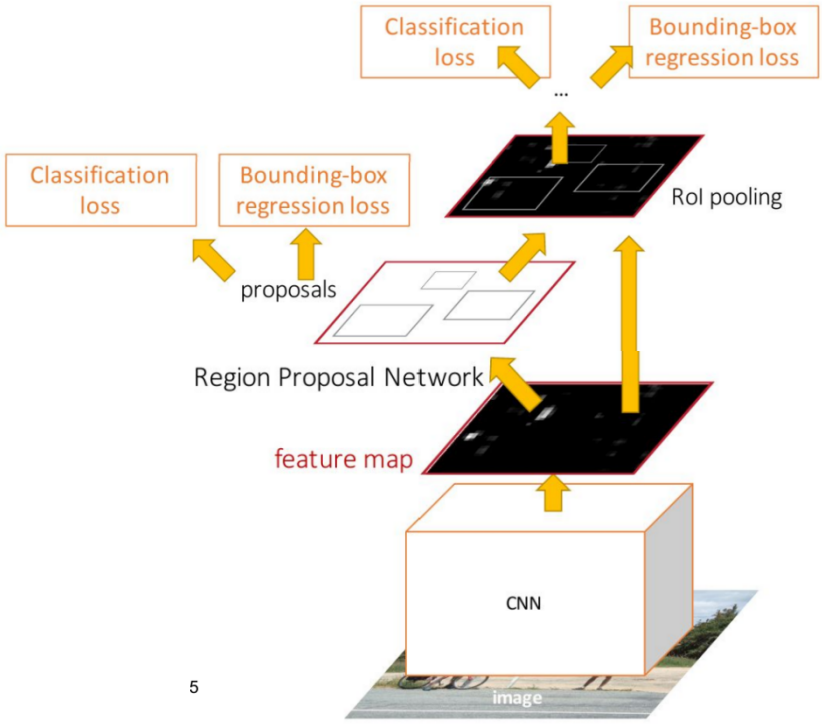
② fully convolutional: use a network with a bunch of convolutional layers, label each activation of the feature maps, and upsampling to the original image. This is because the original image may have a high resolution and convolutional operation can be very expensive.



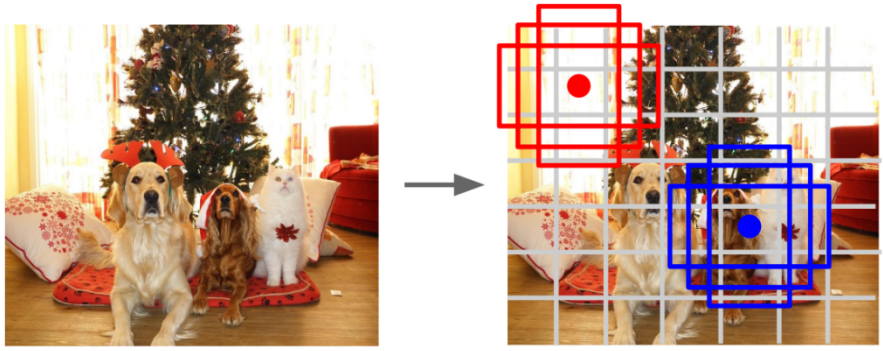
object detection: the goal is draw a bounding box around an object and label it as a certain category.

ideas: ① sliding window: choose a window size, slide across the image, and each portion of image is fed into a CNN model and makes a prediction. After several hundred of iterations, objects have been detected.

② region proposal: R-CNN → Fast R-CNN → Faster R-CNN. The key idea is extract features using CNN, insert a region proposal network to predict several regions, each represented by four numbers (x-coordinate, y-coordinate, height, width), then filter out those undesirable regions, and the rest is fed into the linear classifier to predict label.



③ YOLO: divide image into grids, each of which has many anchor boxes that remain fixed at each grid's center before adjusting each box's positions and scale, represented by 5 numbers (x-coordinate, y-coordinate, height, width, confidence score).



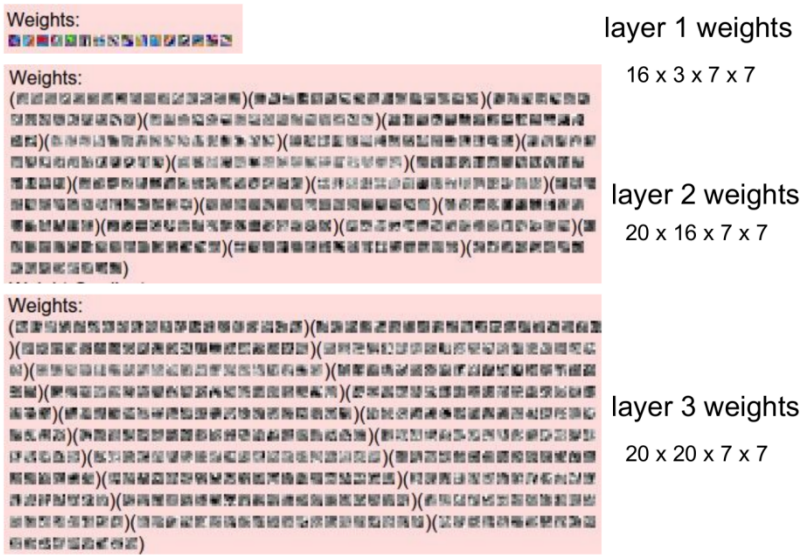
instance segmentation: apart from the localization each object in an image using bounding box, instance segmentation requires us to draw an edge line around each object.

5. Neural Network Visualization

motivation: to understand the underlying behavior of each layer; to visualize which portion of the original image contribute to the final classification; to manipulate the output of activations of feature maps in order to build a fancy model.

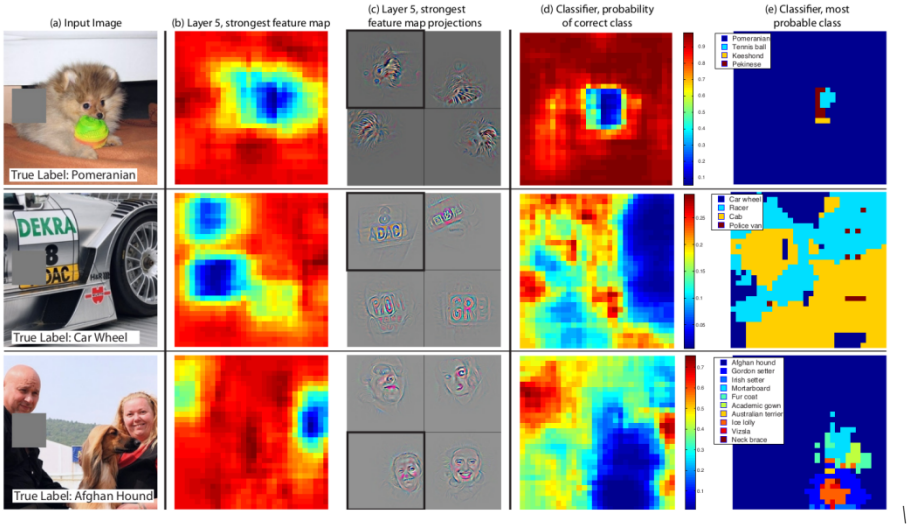
to do: kernel visualization, saliency maps, class visualization, image fooling, neural style transfer.

kernel visualization: to visualize the weights of each kernel(filter) of each layer.



for higher-level layer, each filter is represented by gray scale image, which is not interesting as we expected.

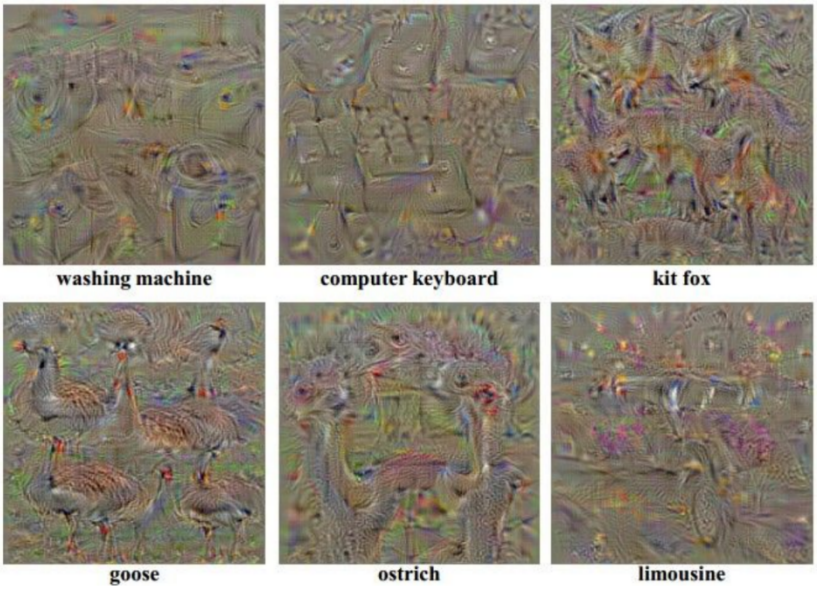
saliency maps: to understand how a model identify an image by masking some portion of the image before feeding to CNN, and draw a heatmap to tell us which part(pixels) matter for classification.



class visualization: choose a target class, we can generate an image that the neural network can recognize as the target class using gradient ascent. Concretely, for an input image I , we obtain the unnormalized class scores$s_y(I)$, then we wish to generate an image from random noise image to maximize that class score, such that:

$$I^* = \arg\max_I s_y(I) - R(I)$$

where $R(I)$ is a regularizer that we want to minimize, with the L_2 regularization form $\lambda\|I\|_2^2$.



application: DeepDream, instead of maximize a specific class score or a neuron using gradient ascent, we amplify the neuron activations at a specific layer in the network.

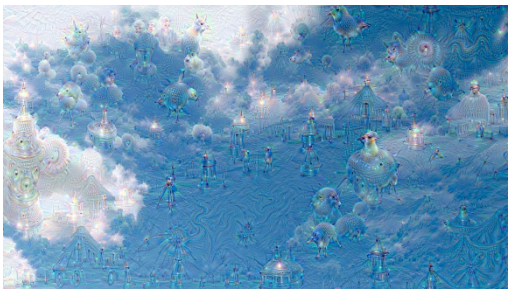
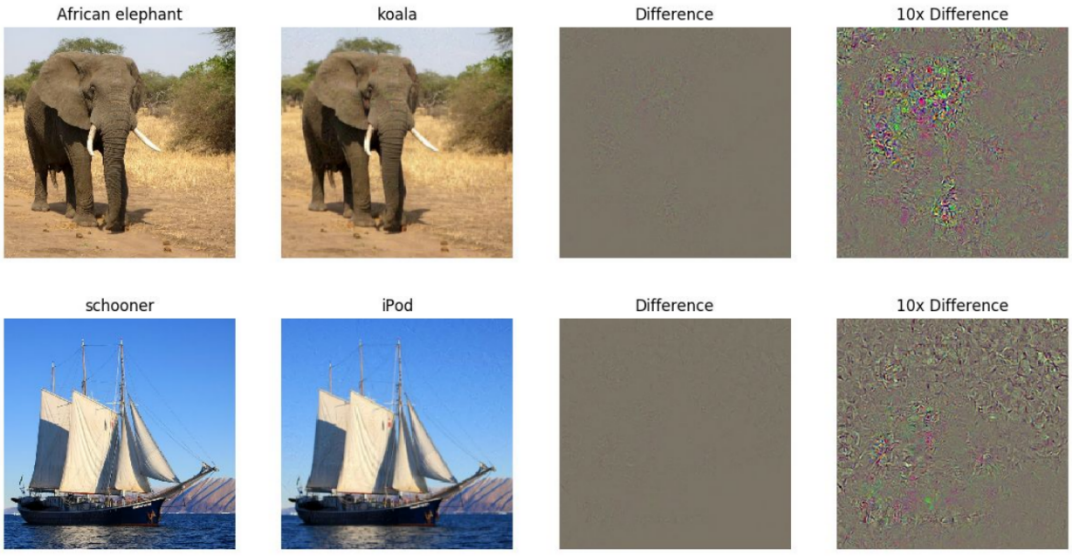


image fooling: given an image and a target class, we perform gradient ascent over the image that maximizes the target class. After that, the generated image fools our model that looks like a certain class by human eyes, but classifies to another class.



neural style transfer: given a content image and a style image, the goal is to generate an image that keeps the content image and reflects the artistic style of the style image. To this end, we start from generating an random

noise image, and tweak every pixels in that image to minimize loss. The total loss contains three components: content loss(evaluate the distance to content image), style loss(evaluate how style-ish of the image) and total variation loss(penalize wiggle and unsmoothness of the image).

content loss:

$$L_c = \omega_c \sum_{i,j} (F_{ij}^{\ell} - P_{ij}^{\ell})^2$$

ω_c is the weight of the content loss in the loss function; F^{ℓ} and P^{ℓ} are squeezed feature map at layer ℓ, which has the size $(N^{\ell} \times M^{\ell})$, where $M^{\ell} = H^{\ell} \times W^{\ell} \times C^{\ell}$.

style loss: to evaluate style loss, the gram matrix is introduced to reflect the approximate covariance of a given image. Given a feature map F^{ℓ} with the shape (C^{ℓ}, M^{ℓ}), the gram matrix G^{ℓ} has the shape (C^{ℓ}, C^{ℓ}).

$$G_{ij}^{\ell} = F^{\ell} F^{\ell^T}$$

For the source style image, we compute the gram matrix A_{ij}^{ℓ}, so the sum of squares of differences of each entry of gram matrix evaluates the style loss of two images.

$$L_s^{\ell} = \omega_s^{\ell} \sum_{i,j} (G_{ij}^{\ell} - A_{ij}^{\ell})^2$$

where ω_s is the weight of the style loss in the loss function. Typically, we usually compute multiple layer's style loss, so we have:

$$L_s = \sum_{\ell \in \mathcal{L}} L_s^{\ell}$$

total variation loss: for a given image with 3 channels(RGB), we compute the sum of squares of differences of all pairs of pixels that are adjacent with each other(horizontally and vertically).

$$L_{tv} = \omega_{tv} \sum_{c=1}^{3} \sum_{i=1}^{H-1} \sum_{j}^{W-1} ((I_{i,j,c} - I_{i+1,j,c})^2 + (I_{i,j,c} - I_{i,j+1,c})^2)$$

where ω_{tv} is the weight of the total variation loss in the loss function.

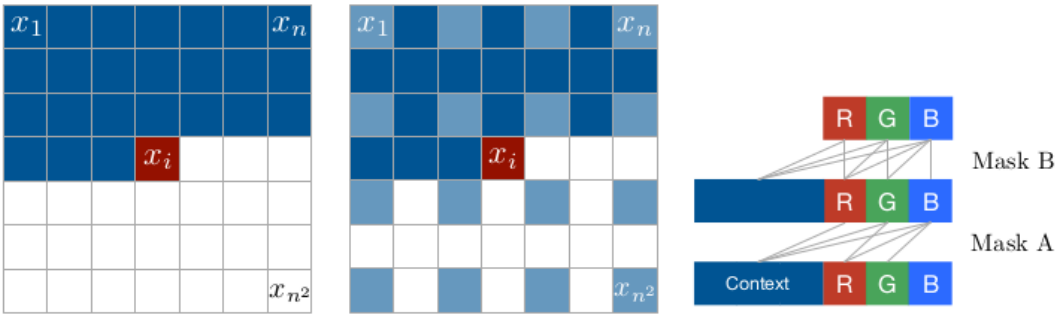
6. Generative Model

motivation: So far, the model we use is always a discriminative that any input is corresponding a label output(classification labels, localization labels, etc.), so what we want is to generate an image given by some distribution of the training set.

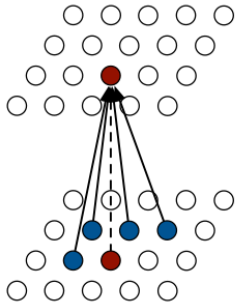
models: PixelRNN, PixelCNN, variational autoencoder, GAN

PixelRNN: since we have already known the distribution of training image, the image can be generated pixel by pixel. The idea is given the previously generated pixel values $\{x_1 x_2, \cdots, x_{i-1}\}$, we predict probability of i'th pixel value x_i by the following formula:

$$p(\mathbf{x}) = \prod_{i=1}^{n^2} p(x_i | x_1, ..., x_{i-1})$$



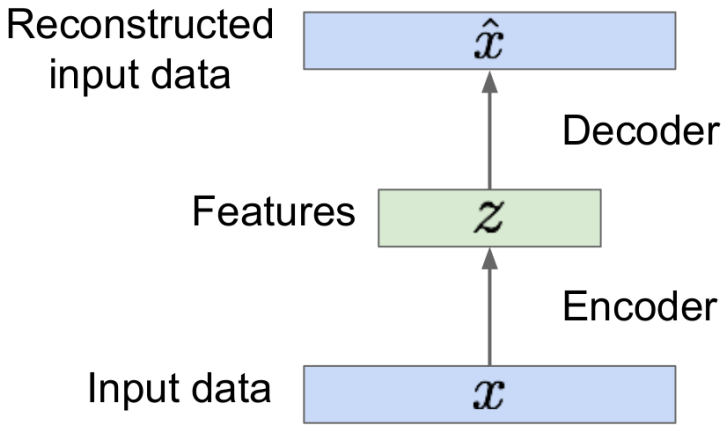
PixelCNN: In PixelRNN, every pixel generated is dependent on all previous pixels, which requires our model has a very long sequence. Therefore, pixels generated by PixelCNN are only dependant on the pixels within the receptive field, in which any future pixels will be masked to 0 during training.



PixelCNN

variational autoencoder: similar to the process of zipping a large file by web server when we download it, and unzip these zipped files in our local machine, the variational autoencoder extract features from the original data and then reconstruct them, which, in turns, has no labels.

naive autoencoder: given an input data *x*, the encoder extracts from the original data to feature *z*, done by a neural network, then the decoder tries to reconstruct back to original data $\tilde{x}$ performed by another neural network.

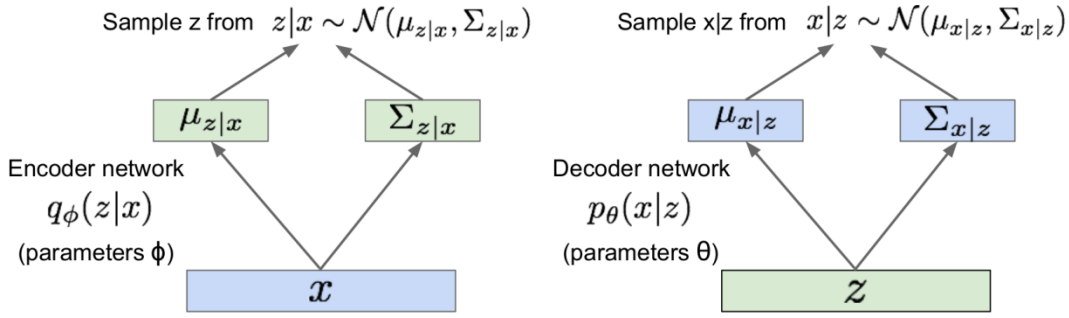


the differences between reconstructed input data and original data is evaluated by L_2 loss function:

$\|x - \hat{x}\|^2$, so the learning objective is to minimize L_2 loss function, and after a several hundred iterations, the

reconstructed data looks similar to the original ones. However the naive autoencoder only generates image that in the training set, while those not in the training set(e.g. look like either two images) will not be generated. Thus, a variant of autoencoder(VAE) will substitute our naive model.

variational autoencoder: rather than obtaining the exact feature values z, we compute the distribution of features of z. In particular, the mean and variance of z given x is the encoded data, so what the decoder will do is sample from that distribution $\mathcal{N}(\mu_{z|x}, \sum_{z|x})$ and reconstruct generated data $\tilde{x}$.



generative adversarial network(GAN): So far, our generative model is sampled from a complex training distribution, which has no direct way to do this. Thus, we turn our gear to sample from a simple distribution, say, random noise, and transform them into a complex image by a neural network.

In GAN, there are two networks, one is a classification network called **discriminator**, which discriminates between the real image in the training set and fake image we generate, another one is called **generator**, which is a complex transformation network that generates a fake image to fool the discriminator into classifying the fake image to a real label.

loss function: the loss function of GAN has two parts, one is for the discriminator, which evaluates the performance of discrimination between a real image and fake image, another is for generator, which evaluates the scale of fake image fooling the discriminator. So we have a min-max objective function.

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Numerically, the objective of discriminator is let probability score for a real image $D_{\theta_d}(x)$ close to 1 and for a fake image $D_{\theta_d}(G_{\theta_g}(z))$ close to 0, whereas the objective of generator is encourage the probability of a fake image being real close to 1, thus we can apply gradient ascent on discriminator and gradient descent on generator. However, training the generator at the beginning does not work well, since when $D_{\theta_d}(G_{\theta_g}(z))$is close to 0, the loss function for generator $\mathbb{E}_{z\sim p(z)}\log(1 - D_{\theta_d}(G_{\theta_g}(z)))$ obtains very low gradient during backpropagation. So what we need to do is change the generator loss function to $\mathbb{E}_{z\sim p(z)}\log(D_{\theta_d}(G_{\theta_g}(z)))$, which means the probability of a fake image being fake and the objective is to maximize this loss function.