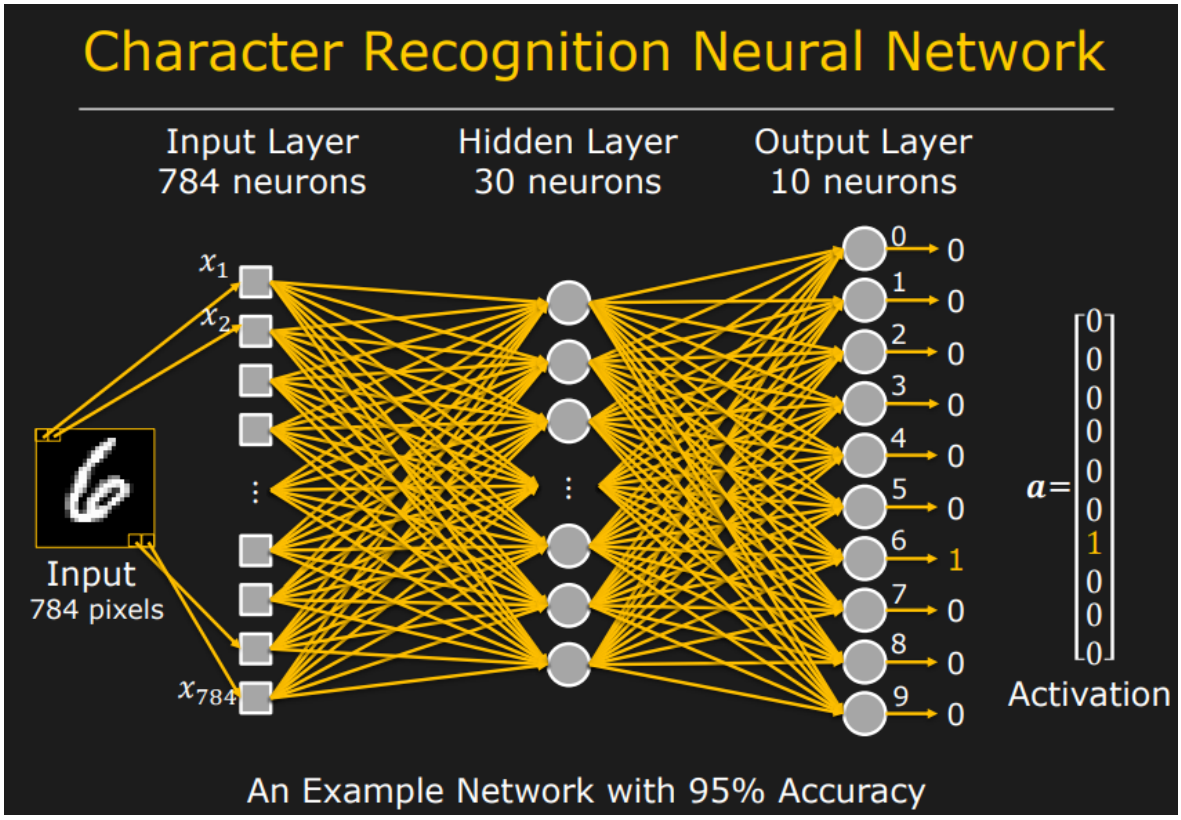
Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Deep learning Book

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Introduction to deep learning



Challenges in image recognition: viewpoint variation, scale variation, deformation, occlusion, intra-class variation. A good image classification model must be invariant to the cross product of all these variations, while simultaneously retaining sensitivity to the inter-class variations.

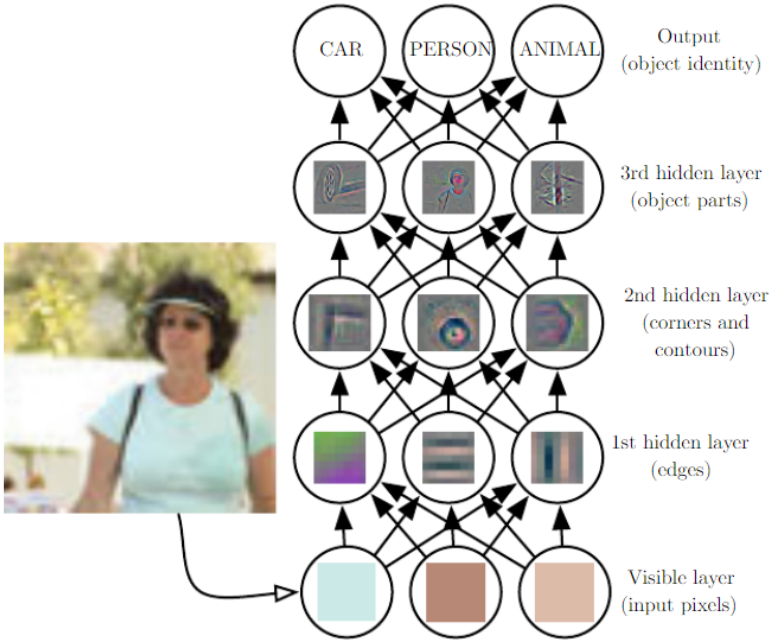
Data driven approach - develop learning algorithms that look at the training data and learn about the visual appearance of each class.

- Nearest Neighbor classifier – take a test image and compare it to every single one of the training images, and predict the label of the closest training image. The closest training image can be chosen using L1 or L2 distance.
- K nearest neighbor - instead of finding the single closest image in the training set, we will find the top k closest images, and have them vote on the label of the test image. In particular, when k = 1, we recover the Nearest Neighbor classifier. Intuitively, higher values of k have a smoothing effect that makes the classifier more resistant to outliers.

Problem of Nearest neighbor: there is no training phase, but we pay computational cost in test time. Another problem is that comparing the distance between two images that contain many pixels and distances over high dimensional spaces can be very counter intuitive – the same two images, when shifted a bit from the other will get a high L1 and L2 distance.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Most of the real world artificial intelligence applications require us to disentangle (separate) the factors of variation (position of the color, the color of the cae, the angle of the car, brightness..) and discard the ones that we do not care about. Deep learning solves this central problem in representation learning by introducing representations that are expressed in terms of other, simpler representations.



Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Data Preprocessing

For all the normalization techniques, the mean must be computed only over the training data and then subtracted equally from all splits (train/val/test).

1) Mean subtraction (zero-centering)

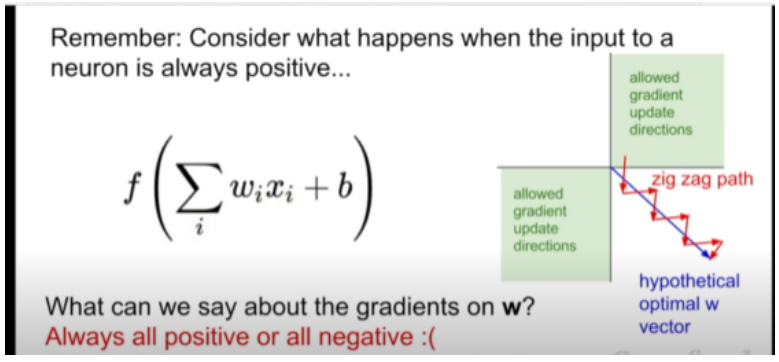
Is the most common form of preprocessing . It involves subtracting the mean across every individual feature in the data, and has the geometric interpretation of centering the cloud of data around the origin along every dimension. In numpy, this operation would be implemented as: X -= np.mean(X, axis = 0). With images specifically, for convenience it can be common to subtract a single value from all pixels (e.g. X -= np.mean(X)), or to do so separately across the three color channels. More explanation: compute the mean matrix of all the training data – for each pixel compute the mean across all the training data, resulting in “mean image”.then subtract from each image the mean matrix .

- We can also subtruct per channel mean (e.g VGG). Instead of having entire mean image that were going to zero-center by, we just take the mean by channel (for each channel, compute the mean of all the images)

In the test, we apply this exact same mean image computed on training images to the test data.

Why do we always want zero-mean data?

- The reason is that if all the data is positive, the gradients in the network (gradient on W) will be all positive or all negative. The gradients with respect to the weights are the gradients of the loss * the gradients of the function of the network w.r.t to w. The gradient of the loss is scalar and with sigmoid, the gradients of the function is x (that is all positive for example, if not normalizing). This means that the gradients of the weight can be **all** negative or all positive, and the update of the weights will be in zig zag. (the weights of the node can either all increase or all decrease at the same time in a single step of gradient descent. It is simply not possible for some weights of the node to increase while some others decrease)



Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

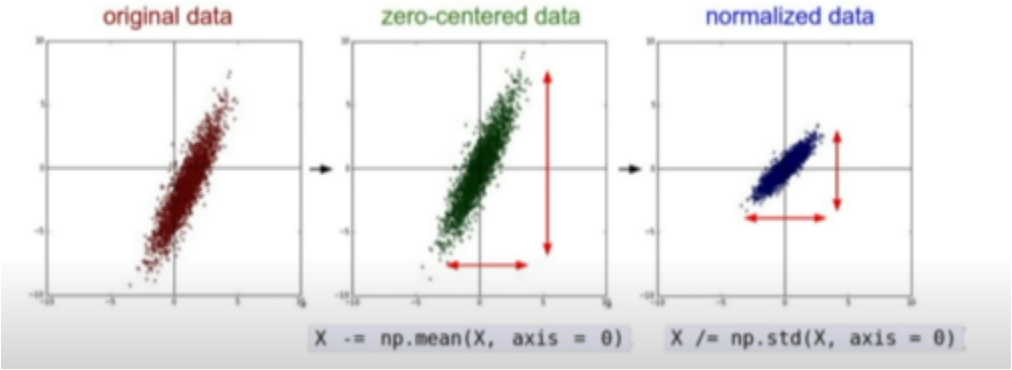
2) Normalization

Refers to normalizing the data dimensions so that they are of approximately the same scale. We treat each pixel as a feature. We would like for each feature to have a similar range so our gradients don't go out of control, and each one will contribute equally. (the second reason was described above in "Why do we always want zero mean data?").

There are two common ways of achieving this normalization.

1) normalizing in the range [-1,1]: divide each dimension by its standard deviation, once it has been zero-centered.
   X -= np.mean (X, axis=0)
   X /= np.std(X, axis = 0))
2) normalizing in the range [0,1] -> less recommended.
   x = (x-min(X))/ (max(X) - min(X))

It only makes sense to apply this preprocessing if you have a reason to believe that different input features have different scales (or units), but they should be of approximately equal importance to the learning algorithm. In the case of images, the relative scales of pixels are already approximately equal (and in range from 0 to 255), so it is not strictly necessary to perform this additional preprocessing step.



3) **PCA** - The data is first centered.Then, we can compute the covariance matrix that tells us about the correlation structure in the data. We can use this to reduce the dimensionality of the data by only using the top few eigenvectors, and discarding the dimensions along which the data has no variance. This is also sometimes referred to as Principal Component Analysis (PCA) dimensionality reduction.It is very often the case that you can get very good performance by training linear classifiers or neural networks on the PCA-reduced datasets, obtaining savings in both space and time.
4) Whitening - after performing PCA, each dimension is additionally scaled by the eigenvalues.

Weight initialization

- Mistake - initialize all the weights to zero. why? Because if every neuron in the network computes the same output, then they will also all compute the same gradients during backpropagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.

- First idea: initialize the weights of the neurons to small random numbers and refer to doing so as symmetry breaking. The idea is that the neurons are all random and unique in the beginning, so they will compute distinct updates and integrate themselves as diverse parts of the full network. It works okay for small networks, but there is going to be problems with deeper networks

  problem: at the beginning the mean and variance is good, but as long as we go deeper in the network the variance shrinks and goes to zero - when we multiply again and again small numbers, at the end we get zero.

- If we initialize the weights with big numbers it will cause problems when using tanh or sigmoid (the gradient will be zero and the weights will not update)
- Xavier initialization
  we want the variance of the input (fan-in) to be the same as the variance of the output (fan-out). If we have small number of inputs, we divide by smaller number and get larger weights (we need larger weight to each input)

  It turns out that we can normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its fan-in (i.e. its number of inputs). That is, the recommended heuristic is to initialize each neuron's weight vector as: w = np.random.randn(fan-in, fan-out) / sqrt(n), where n is the number of its inputs. This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence**.**

  The problem with Xavier is when using ReLU since ReLU kills approximately half of the neurons and the assumptions of fan-in and fun-out do not hold.

  A more recent paper on this topic, derives an initialization specifically for ReLU neurons, reaching the conclusion that the variance of neurons in the network should be 2.0/n. Now we are adjusting to the fact that half of the neurons get killed. This gives the initialization w = np.random.randn(n) * sqrt(2.0/n), and is the current recommendation for use in practice in the specific case of neural networks with ReLU neurons.

Activation functions

An activation function is a function that is added into an artificial neural network in order to help the network learn complex patterns in the data. It takes in the output signal from the previous cell and converts it into some form that can be taken as input to the next cell.
An activation function in a neural network defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network. All hidden layers typically use the same activation function. The output layer will typically use a different activation function from the hidden layers and is dependent upon the type of prediction required by the model.

There are multiple reasons for having non-linear activation functions in a network.

1) They also help in keeping the value of the output from the neuron restricted to a certain limit as per our requirement. This is important because input into the activation function is W*x + b where W is the weights of the cell and the x is the inputs and then there is the bias b added to that. This value if not restricted to a certain limit can go very high in magnitude especially in the case of very deep neural

networks that have millions of parameters. This will lead to computational issues. For example, there are some activation functions (like softmax) that output specific values for different values of input (0 or 1).

2) The most important feature in an activation function is its ability to add non-linearity into a neural network. Without activation functions all of our layers would simply stack one affine (product plus addition) transformation after another. Each layer would simply add a vector product, and vector addition, to the previous one. It can be shown that this composition of affine transformations is equivalent to a single affine transformation. There are many problems a linear transformation can't solve, so we would effectively be shrinking the quantity of functions our model could estimate.

Activation for hidden layers

A hidden layer in a neural network is a layer that receives input from another layer (such as another hidden layer of an input layer) and provides output to another layer (such as another hidden layer or an output layer).

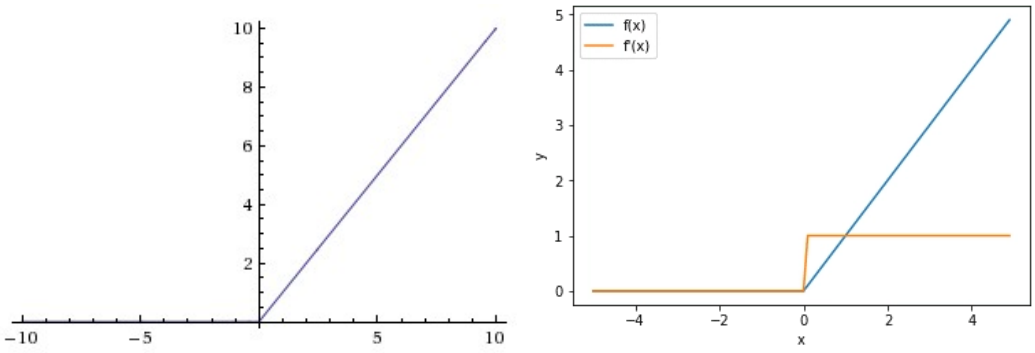
There are perhaps three activation functions you may want to consider for use in hidden layers:

- Rectified Linear Activation (ReLU)
- Logistic (Sigmoid)
- Hyperbolic Tangent (Tanh)

1) ReLU

The most common function that is used for hidden layers. Specifically, it is less susceptible to vanishing gradients that prevent deep models from being trained, although it can suffer from other problems like saturated or "dead" units.

$$Relu(x) = Max(0, x)$$



the activation is simply thresholded at zero. It was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.

The advantages of ReLU compared to sigmoid/tanh –

The first derivative of ReLU is equal to 1 when x is greater than zero, but otherwise it is 0 .When it's derivative is back-propagated there will be no degradation of the error signal as 1 x 1 x 1 x 1… = 1. However, the ReLU activation still maintains a non-linearity or "switch on" characteristic which enables it to behave analogously to a biological neuron.

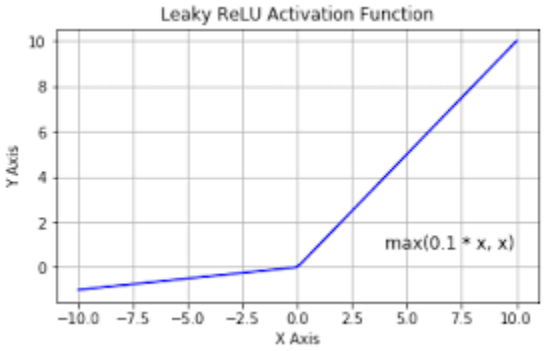
Unfortunately, ReLU units can be fragile during training and can “die”. For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any data point again. In order to encourage neurons to be active, people like to initialize ReLU neurons with slightly positive biases (e.g 0.01)

- When using the ReLU function for hidden layers, it is a good practice to use a “He Normal” or “He Uniform” weight initialization and scale input data to the range 0-1 (normalize) prior to training.

2) LeakyRelu

Leaky ReLUs are one attempt to fix the “dying ReLU” problem. The problem is that ReLU outputs zeros for any negative value. If the network’s weights reach such values that they always yield negative values when multiplied with the inputs, then the entire ReLU-activated unit keeps producing zeros (this is the reason we should initialize weights with range (0,1)). If many neurons die like this, the network's learning capabilities get impaired.

Instead of the function being zero when x < 0, a leaky ReLU will instead have a small negative slope (of 0.01, or so).

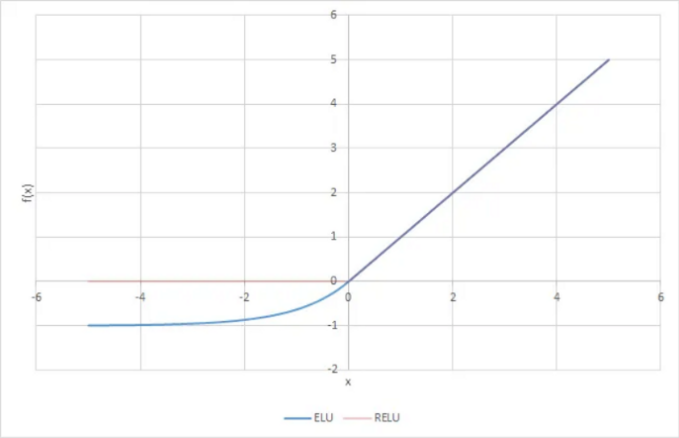


3) ELU

$$\mathrm{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases}$$

If you input an x-value that is greater than zero, then it's the same as the ReLU – the result will be a y-value equal to the x-value. But this time, if the input value x is less than 0, we get a value slightly below zero. Advantage: ELU is smooth around zero, which speeds up convergence.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



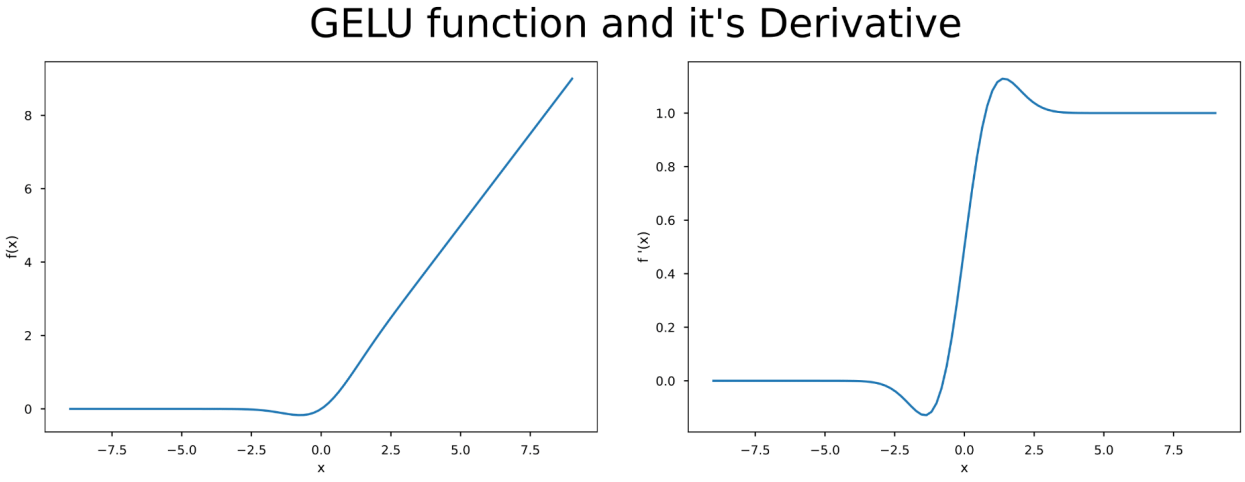
4) GeLu

Gaussian Error Linear Unit. An activation function used in GPT and BERT.

$$gelu(x) = xP(X \leq x) \qquad X \sim \mathcal{N}(0,1)$$

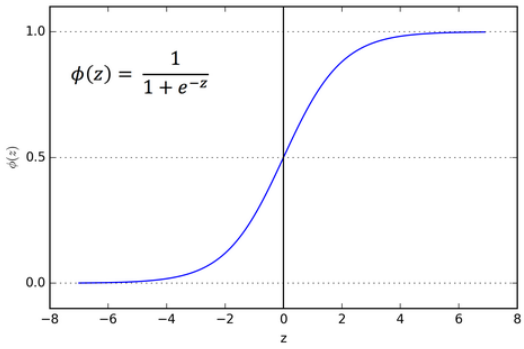
where x is an input value, X is a Gaussian random variable with zero-mean and unit-variance that models the input, and P(X≤x) is the probability that the input can be less than or equal to the given value.

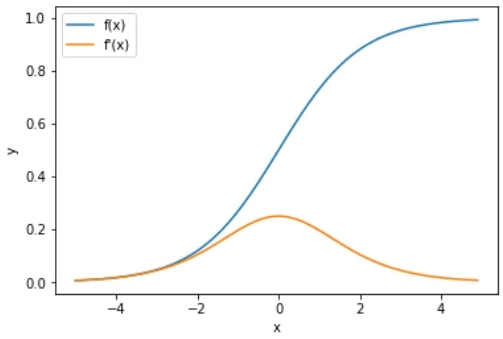
$$\text{GELU}(x) = 0.5x\left(1+\tanh\left(\sqrt{2/\pi}(x+0.044715x^3)\right)\right)$$



Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

5) sigmoid





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

The sigmoid function takes a real-valued number and "squashes" it into a range between 0 and 1. In particular, large negative numbers become 0 and large positive numbers become 1. It is especially used for models where we have to predict the probability as an output.
The problem of sigmoid - **saturation and killing gradients.** A very undesirable property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. As can be observed from the top right image, when the sigmoid function value is either too high or too low, the derivative (orange line) becomes very small i.e. << 1. This causes vanishing gradients and poor learning for deep networks. This can occur when the weights of our networks are initialized poorly – with too-large negative and positive values.

- When using the Sigmoid function for hidden layers, it is a good practice to use a "Xavier Normal" or "Xavier Uniform" weight initialization (also referred to Glorot initialization, named for Xavier Glorot) and scale input data to the range 0-1 (e.g. the range of the activation function) prior to training.

6) Softmax

The **softmax** function is a more generalized logistic activation function that is used for multiclass classification.

The standard (unit) softmax function $\sigma : \mathbb{R}^K \to [0,1]^K$ is defined by the formula

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K.$$

In simple words, it applies the standard exponential function to each element z_i of the input vector $\mathbf{z}$ and normalizes these values by dividing by the sum of all these exponentials; this normalization ensures that the sum of the components of the output vector $\sigma(\mathbf{z})$ is 1.

underflow - when numbers near zero are rounded to zero (when? dividing by zero, taking the logarithm of zero..)
overflow - numbers with large magnitude.
Softmax function must be stabilized against underflow and overflow.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

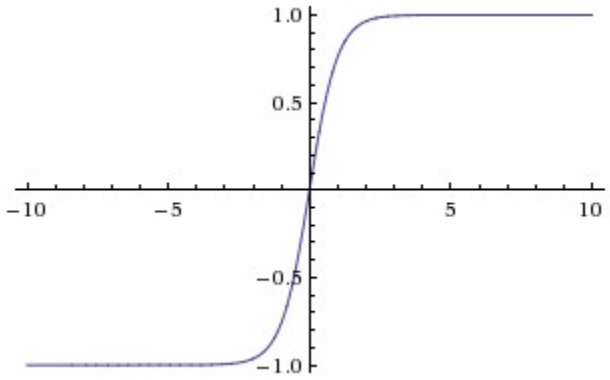
Consider what happens when all the z_i equal to some constant c.

Analytically, we can see that all the outputs should be equal to 1/n. Numerically, this may not occur when *c* has a large magnitude:

1) If c is very negative, then exp(c) will underflow. This means the denominator (מכנה) of the softmax will become 0, so the final result is undefined.
2) When c is very large and positive, exp(c) will overflow, again resulting in the expression as a whole being undefined.

Both of these difficultiescan be resolved by instead evaluating softmax(x) where $x = z - max_i z_i$. Simple algebra shows that the value of the softmax function is not changed analytically by adding or subtracting a scalar from the input vector. Subtracting $max_i z_i$ results in the largest argument to exp being 0, which rules out the possibility of overflow. Likewise, at least one term in the denominator has a value of 1, which rules out the possibility of underflow in the denominator leading to a division by zero.

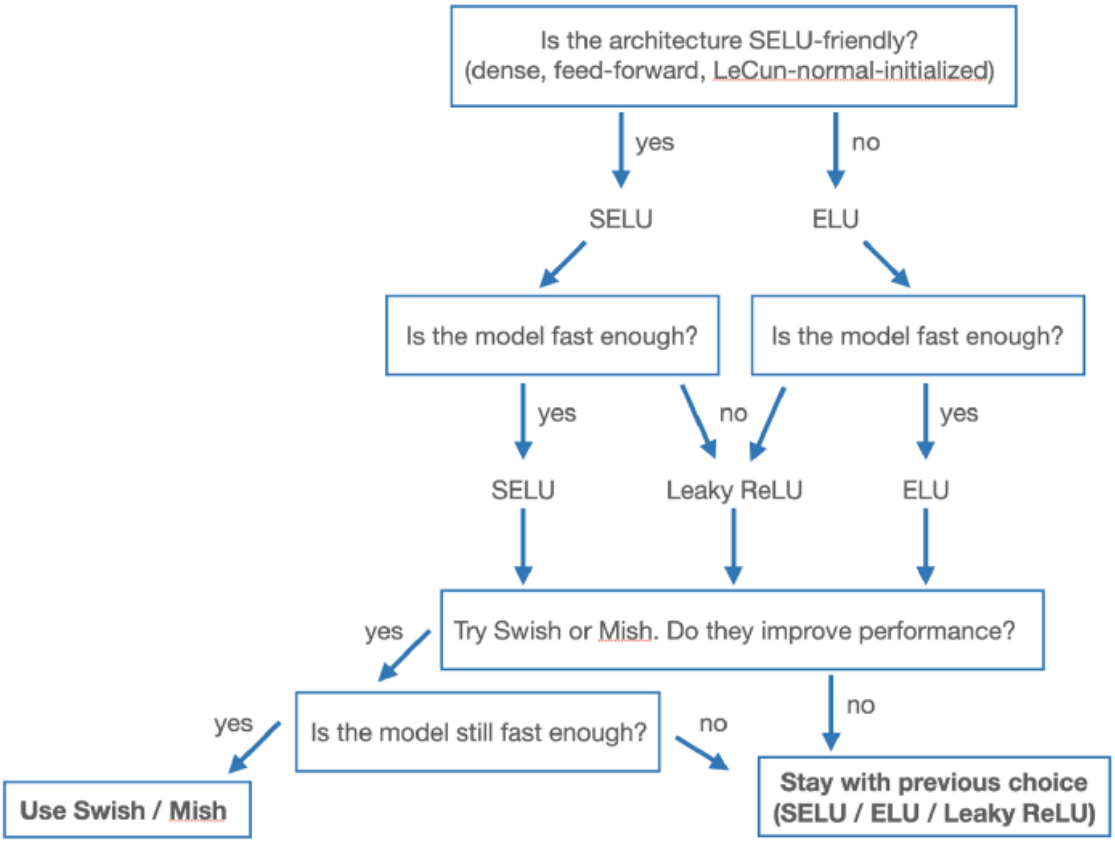
7) Tanh



tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1).

The tanh squashes a real-valued number to the range [-1, 1]. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron, its output is zero-centered. Therefore, in practice, the tanh non-linearity is always preferred to the sigmoid nonlinearity.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Loss functions

1) L1 distance (Mean Absolute Error loss) - Regression

On some regression problems, the distribution of the target variable may be mostly Gaussian but may have outliers, e.g. large or small values far from the mean value. The Mean Absolute Error, or MAE loss is an appropriate loss function in this case as it is more robust to outliers. It is calculated as the average of the absolute difference between the actual and predicted values - we should convert both images to vectors and sum the difference between every two coordinates

$$MAE\ (I1,\ I2)\ =\ \sum_{p} |\ I_1^{\ p} -\ I_2^{\ p}\ |$$

2) L2 distance (MSE loss) - Regression

Euclidean distance between two vectors.

$$MSE(x, y)\ =\ L\ =\ \{l_1,\ ..,\ l_N\}^T,\ l_n\ =\ (x_n -\ y_n)^2$$

where N is the batch size.

L2 distance is much more unforgiving than L1 distance when it comes to differences between two vectors. That is, the L2 distance prefers many medium disagreements to one big one.

- Generally, L2 loss converges faster than L1. However, L1 seems to be better than MSE in image generation tasks such as super-resolution and image distortion tasks.
- **What is the problem with L2 loss in image distortion tasks?**

when optimizing an image restoration model using MSE loss, the resulting images are blurry. The problem is in the shape of the loss as it approaches zero. Explanation - suppose we are minimizing the MSE loss between the original image (target) and the image output (the clean image). As we keep training the network, the error becomes closer to zero. MSE loss is a parabola meaning its derivative function is $2(x_n -\ y_n)$. Thus, the closer the error is to zero, the smaller the gradient is, meaning that small deviation from the ground truth is not penalized as much. (and the small deviations are the fine details which are important to sharpness). In contrast, L1 has constant gradients, which means that with the loss approaching zero, the gradient will not diminish, resulting in sharper-looking images.

3) Binary cross entropy - Binary classification

The binary cross-entropy is appropriate in binary classification settings to get one of two potential outcomes. The loss is calculated according to the following formula, where y represents the expected outcome, and y hat represents the outcome produced by our model.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

$$L = -(y_i(log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i))$$

remember: $log(1) = 0$, $log(0) = -\inf$ so if the target y_i is 1, and the output $\hat{y}_i$ is 1, log(1)=0 thus the loss will be 0. In contrast, if the target is 1 and the output is 0, log(0) = -inf, and the loss will be inf.

4) Cross-Entropy - multiclass classification

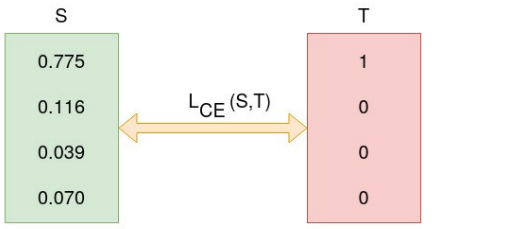
applied when using softmax classifier, i.e. when the output is normalized class probabilities (the probability should be close to 1 just for the target class).

Given that M is the number of classes, the formula is as follows.

$$L = -\sum_{j=1}^{M} y_j log(\hat{y}_j)$$

As the probability for the correct class is larger, the log($\hat{y}_i$) is larger, and the - log($\hat{y}_i$) is smaller.

Note that the cross entropy loss rewards/penalises probabilities of correct classes only. The value is independent of how the remaining probability is split between incorrect classes.
example:



Logits(S) and one-hot encoded truth label(T) with Categorical Cross-Entropy loss function used to measure the 'distance' between the predicted probabilities and the truth labels. (Source: Author)

The categorical cross-entropy is computed as follows

$$\begin{aligned} L_{CE} &= -\sum_{i=1} T_i \log(S_i) \\ &= -[1\log_2(0.775) + 0\log_2(0.126) + 0\log_2(0.039) + 0\log_2(0.070)] \\ &= -\log_2(0.775) \\ &= 0.3677 \end{aligned}$$

Note in PyTorch:

If we have M classes the output should be of shape (N,M) where N is the batch size and the target will be of shape (N) where each value represents a number from 0,1...M.
For example, I have 5 classes and an image with target label 2. thus the ideal predicted output will be (0,1,0,0,0) and the target is 1.

nn.crossentorpyloss() combines softmax and NLL loss so we need to pass the logits as they are.

5) Mean Squared Logarithmic Error Loss

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(\log(x_i) - \log(y_i))^2}.$$

There may be regression problems in which the target value has a large range of values and when predicting a large value, you may not want to punish a model as heavily as mean squared error. Instead, you can first calculate the natural logarithm of each of the predicted values, then calculate the mean squared error. It has the effect of relaxing the punishing effect of large differences in large predicted values. As a loss measure, it may be more appropriate when the model is predicting unscaled quantities directly.

6) KL divergence

Kullback Leibler Divergence, or KL Divergence for short, is a measure of how one probability distribution differs from a baseline distribution.
A KL divergence loss of 0 suggests the distributions are identical. In practice, the behavior of KL Divergence is very similar to cross-entropy. It calculates how much information is lost (in terms of bits) if the predicted probability distribution is used to approximate the desired target probability distribution.
As such, the KL divergence loss function is more commonly used when using models that learn to approximate a more complex function than simply multi-class classification, such as in the case of an autoencoder used for learning a dense feature representation under a model that must reconstruct the original input. In this case, KL divergence loss would be preferred. Nevertheless, it can be used for multi-class classification, in which case it is functionally equivalent to multi-class cross-entropy.

Entropy - in general, it gives the average amount of information that we get from one sample drawn from a given probability distribution p. It tells us how unpredictable the probability distribution is.

high entropy - more “surprise” (uniform distribution)

low entropy - one-hot vector.

When the predicted distribution is equal to the true distribution, then the cross-entropy is simply equal to entropy. But, if the distributions differ, then the cross-entropy will be greater than the entropy by some number of bits. This amount by which the cross-entropy exceeds the entropy is called the Relative Entropy or more commonly known as the Kullback-Leibler Divergence (KL Divergence).

$$CrossEntropy = Entropy + KL - divergence$$

Usually, in most deep learning tasks, The “Entropy” is constant. This entropy here represents the entropy of the training dataset, which is known and not learned by the model. So, only the KL divergence is important.

The only case where the KL divergence is not equal to the cross-entropy is when the “target distribution” is also learned (as in VAE).

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

7) Perceptual loss

Perceptual loss functions are used when comparing two different images that look similar, like the same photo but shifted by one pixel. The function is used to compare high-level differences, like content and style discrepancies, between images. A perceptual loss function is very similar to the per-pixel loss function (L1 loss), as both are used for training feed-forward neural networks for image transformation tasks. The perceptual loss function is a more commonly used component as it often provides more accurate results regarding style transfer.

Basically (in high level), we take the output image $\hat{y}$ and the target image y and pass them through the pretrained network. For both of the images, the output of the pretrained network is a feature map. So after this stage, we get two feature maps, one for the output $\phi(\hat{y})$ and the other for the target $\varphi(y)$. Rather than telling our transformed network output(y=f(x)) to match up exactly to y(our output image) pixel wise, we encourage them to have similar feature representation, rather than pixels. The feature reconstruction loss is the L2 distance between the both feature maps.

- Drawbacks: Feature-based losses have multiple drawbacks — they are computationally expensive, require regularization and hyper-parameter tuning, involve a large network trained on an unrelated task, and thus the training process for the image restoration task is very memory intensive.

8) Distribution loss/Adversarial loss (using GAN)

used for image restoration tasks with more than one acceptable solution. For example, images produced by super-resolution or denoising algorithms can have acceptable perceptual quality while not precisely matching the ground truth.
In such a setting, the image-generation algorithm has several loss terms: the discriminator, trained to differentiate between the generated and natural images, and one or several loss terms (as L1) constraining the generator network to produce images close to the ground truth.

9) IoU loss - object detection

```
def bb_intersection_over_union(boxA, boxB):
	# determine the (x, y)-coordinates of the intersection rectangle
	xA = max(boxA[0], boxB[0])
	yA = max(boxA[1], boxB[1])
	xB = min(boxA[2], boxB[2])
	yB = min(boxA[3], boxB[3])
	# compute the area of intersection rectangle
	interArea = max(0, xB - xA + 1) * max(0, yB - yA + 1)
	# compute the area of both the prediction and ground-truth rectangles
	boxAArea = (boxA[2] - boxA[0] + 1) * (boxA[3] - boxA[1] + 1)
	boxBArea = (boxB[2] - boxB[0] + 1) * (boxB[3] - boxB[1] + 1)
	# compute the iou by taking the intersection area and dividing it by the sum
	# of prediction + ground-truth areas - the interesection area
	iou = interArea / float(boxAArea + boxBArea - interArea)
	return iou
```

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

10) Style loss

the style loss function makes sure that the correlation of activations in all layers is similar between the style image and the generated image. The style information is measured as the amount of correlation present between features maps in a given layer.

These feature correlations are given by the Gram matrix $G^l \in R^{N_l \times N_l}$, where $G_{ij}^{\ l}$ is the inner product between the vectorized feature map i and j in layer l:

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l.$$

here, we minimize the mean-squared distance between the entries of the Gram matrix from the original image and the Gram matrix of the image to be generated. So let a and x be the original image and the image that is generated and A^l and G^l their respective style representations in layer l. The contribution of that layer to the total loss is then:

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^l - A_{ij}^l\right)^2$$

and the total loss is:

$$\mathcal{L}_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l$$

where w_l is a weight given to each layer during the loss computation.

The Gram matrix essentially captures the "distribution of features" of a set of feature maps in a given layer. By trying to minimize the style loss between two images, you are essentially matching the distribution of features between the two images

11) Identity loss

minimize the cosine similarity between the feature embeddings of the reconstructed image and its source image. The identity loss is specifically designed to assist in the accurate inversion of real images in the facial domain.

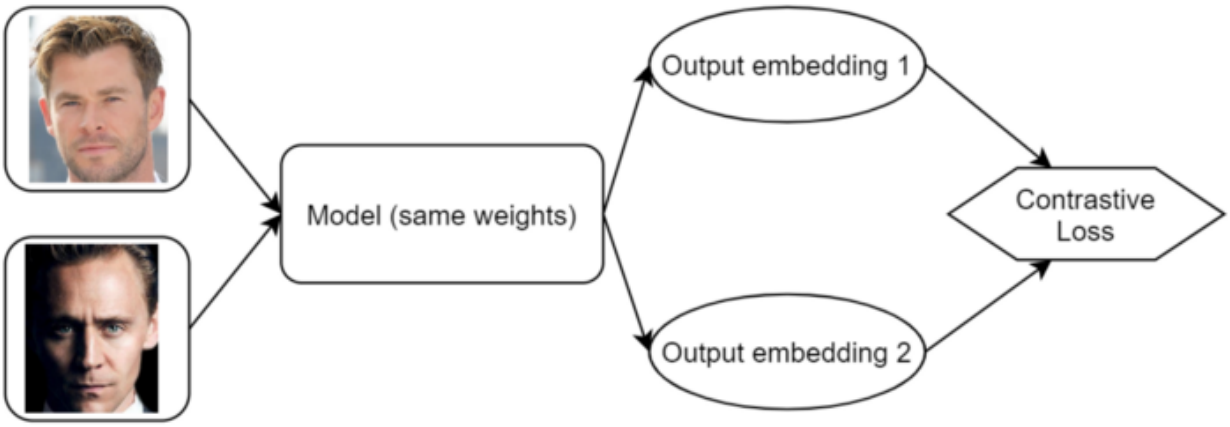
$$\mathcal{L}_{\text{ID}}(w) = 1 - \langle R(G(w_s)), R(G(w)) \rangle,$$

where R is a pertain network (ResNet for example) that extracts features from images.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

12) Contrastive learning loss

When training a Siamese network, 2 or more inputs are encoded and the output features are compared. A siamese network is often shown as two different encoding networks that share weights, but in reality, the same network is just used twice before doing backpropagation.

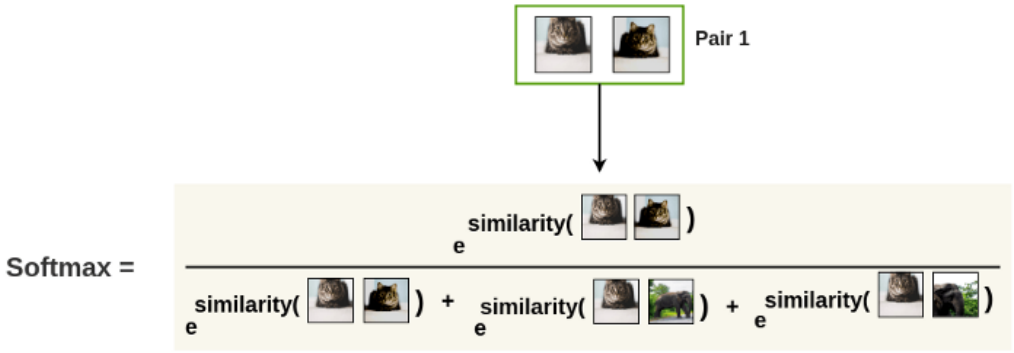


Contrastive loss takes the output of the network for a positive example and calculates its distance to an example of the same class and contrasts that with the distance to negative examples. Said another way, the loss is low if positive samples are encoded to similar (closer) representations and negative examples are encoded to different (farther) representations.

The cosine distance measures the cosine of the angle between the vectors. The cosine of identical vectors is 1 while orthogonal and opposite vectors are 0 and -1 respectively. More similar vectors will result in a larger number. Calculating the cosine distance is done by taking the dot product of the vectors.

$$\ell_{i,j} = -\log \frac{\exp(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_j)/\tau)}{\sum_{k=1}^{2N} \quad _{[k \neq i]} \exp(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_k)/\tau)},$$

the similarity score between image i and image j is computed as follows (in very high level): compute the cosine similarity between the embedding of image$i$ and the embedding of image j, and divide it by the sum of cosine similarities of image i with all the images in the dataset.



13) Hungarian loss for object detection (DETR)

Suppose that the network produces N final predictions $\widehat{y}_1, \ldots \widehat{y}_N$. Each prediction contains a bounding box and class probability. The ground truth contains L elements $y_1, \ldots y_L$. Each element is bounding box + class label.

First, Calculate the best match of predictions with respect to given ground truths using a graph technique with a cost function (matching c ost function). Next, define a loss to penalize the class and box predictions.

matching function -

$$-\mathbb{1}_{\{c_i \neq \varnothing\}} \hat{p}_{\sigma(i)}(c_i) + \mathbb{1}_{\{c_i \neq \varnothing\}} \mathcal{L}_{\text{box}}(b_i, \hat{b}_{\sigma(i)})$$

for each target y_i out of $y_1, \ldots y_L$, the matching function finds the closest prediction $\widehat{y}_j$ from $\widehat{y}_1, \ldots \widehat{y}_N$ outputs of the network. The closest prediction to y_i mean (1) $\widehat{y}_j$ has the highest probability for the class label of i c_i (2) $\widehat{y}_j$'s bounding box is the closest to y_i's bounding box.

Each element i of the ground truth set can be seen as a $y_i = (c_i, b_i)$ where c_i is the target class label (which may be ∅) and $b_i \in [0, 1]$ is a vector that has four attributes — normalized ground truth box center coordinates, height and width relative to the image size. For the prediction with index σ(i) we define the probability of class ci as $\widehat{p}_{\sigma(i)}(c_i)$ and the predicted box as $\widehat{b}_{\sigma(i)}$. The first part of loss takes care of class prediction and the second part is the loss for the box prediction.

box loss function -

$$\lambda_{\text{iou}} \mathcal{L}_{\text{iou}}(b_i, \hat{b}_{\sigma(i)}) + \lambda_{\text{L1}} ||b_i - \hat{b}_{\sigma(i)}||_1 \text{ where } \lambda_{\text{iou}}, \lambda_{\text{L1}} \in \mathbb{R} \text{ are hyperparameters.}$$

Hungarian loss -

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

$$\mathcal{L}_{\text{Hungarian}}(y, \hat{y}) = \sum_{i=1}^{N} \left[-\log \hat{p}_{\hat{\sigma}(i)}(c_i) + \mathbb{1}_{\{c_i \neq \varnothing\}} \mathcal{L}_{\text{box}}(b_i, \hat{b}_{\hat{\sigma}}(i)) \right]$$

$\hat{p}_{\hat{\sigma}(i)}(c_i)$- the probability for the correct class c_i for the prediction number $\hat{\sigma}(i)$ of the model.

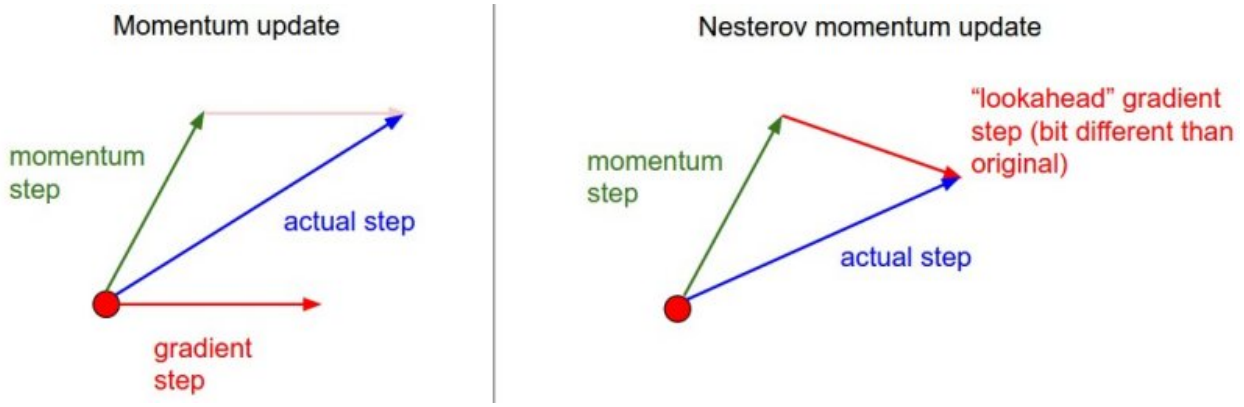
$\hat{b}_{\hat{\sigma}(i)}$ - the predicted box for the prediction number $\hat{\sigma}(i)$.

$\hat{\sigma}$ is the optimal assignment computed in the matching cost function.

Optimization

We can compute the best direction along which we should change our weight vector that is mathematically guaranteed to be the direction of the steepest descent (at least in the limit as the step size goes towards zero). This direction will be related to the gradient of the loss function.

Backpropagation - The forward pass computes values from inputs to output. The backward pass then performs backpropagation which starts at the end and recursively applies the chain rule to compute the gradients all the way to the inputs of the circuit. The gradients can be thought of as flowing backwards through the circuit.



Adagrad - Adaptive learning rate method per parameter (assign to each parameter a different learning rate). Notice that the weights that receive high gradients will have their effective learning rate reduced, while weights that receive small or infrequent updates will have their effective learning rate increased.

Adam - also adaptive learning rate method. similar to adagrad but also uses momentum.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Regularization

There might be many similar W (weights) that correctly classify the examples. one easy way to see this is that if some parameters W correctly classify all examples (so the loss is zero for each example), then any multiple of these parameters λW where λ>1 will also give zero loss. We wish to encode some preference for a certain set of weights W over others to remove this ambiguity. We can do so by extending the loss function with a regularization penalty R(W).

1) L2 regularization - weight decay

The most common regularization penalty is the L2 norm that discourages large weights through an elementwise quadratic penalty over all parameters.

$$Loss\ (x, y)\ =\ Loss\ (x, y)\ +\ \frac{\alpha}{2}\ ||w||^2_2$$

Observation: Penalizing large weights tends to improve generalization because it means that no input dimension (one dimension out of all the dimensions) can have a very large influence on the scores all by itself.
For example, suppose that we have some input vector x=[1,1,1,1] and two weight vectors w_1=[1,0,0,0], $w_2 = [0.25,\ 0.25,\ 0.25,\ 0.25]$. Note that$w_1^T x\ =\ w_2^T x\ =\ 1$so both weight vectors lead to the same dot product. However, the L2 penalty of w_1 is 1.0 while the L2 penalty of w_2 is only 0.5. Therefore, according to the L2 penalty the weight vector w_2 would be preferred since it achieves a lower regularization loss. Since the L2 penalty prefers smaller and more diffuse weight vectors, the final classifier is encouraged to take into account all input dimensions to small amounts rather than a few input dimensions and very strongly.

2) L1 regularization

$$Loss\ (x, y)\ =\ Loss\ (x, y)\ +\ \alpha\ ||w||_1 \quad where\ ||w||_1 = \sum_i |w_i|$$

for each weight w, we add the term λ|w| to the objective. The L1 regularization has the intriguing property that it leads the weight vectors to become sparse during optimization (i.e. very close to exactly zero). In other words, neurons with L1 regularization end up using only a sparse subset of their most important inputs and become nearly invariant to the "noisy" inputs. In comparison, final weight vectors from L2 regularization are usually diffuse, small numbers. In practice, if you are not concerned with explicit feature selection, L2 regularization can be expected to give superior performance over L1.

3) Dropout

While training, dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise. We do not want to use randomness at test time, thus we multiply the output of each neuron by the dropout probability.

By using dropout, the same layer will alter its connectivity and will search for alternative paths to convey the information in the next layer. As a result, each update to a layer during training is performed with a different "view" of the configured layer. Conceptually, it approximates training a large number of neural networks with different architectures in parallel.

This conceptualization suggests that perhaps dropout breaks up situations where network layers co-adapt to correct mistakes from prior layers, making the model more robust. It increases the sparsity of the network and in general, encourages sparse representations!

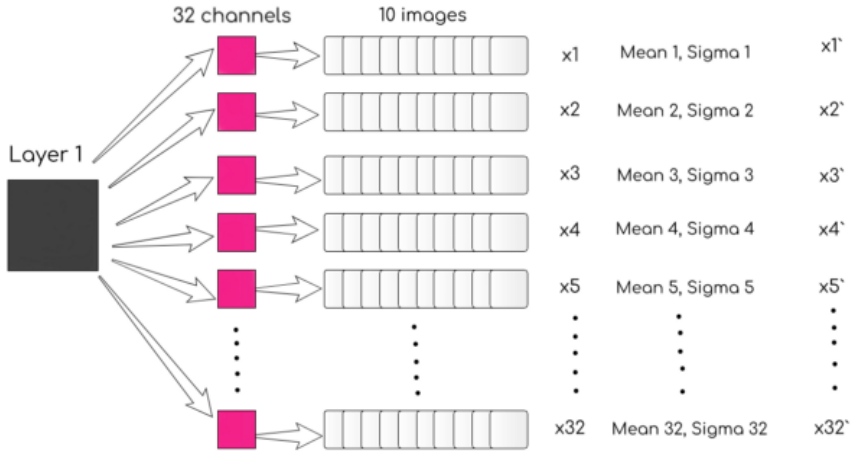
4) Batch Normalization

Batch-Normalization (BN) is an algorithmic method which makes the training of Deep Neural Networks (DNN) **faster** and more **stable**.

In BN, we are taking the mean and the variance of the current batch and normalizing by this. (we calculate it for the batch, for each neuron in the layer of batchnorm). This way, each feature has a mean of 0 and a standard deviation of 1.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Normalizing each intermediate layer's inputs with the dataset mean and standard deviation.
The key thing to note is that normalization happens for all input dimensions in the batch separately (in convolution terms, think channels). the number of means & variances is the same as the number of channels. For example: say we have batch size = 10 (10 images), and we pass the 10 images through the conv layer with 32 output layers. It means that for each origin image there are now 32 feature maps. we do the BN across each feature map - for each feature map we take the mean and variance of all the 10 images:



Usually, we add the BatchNorm layer immediately after fully connected layers (or convolutional layers), and before non-linearities. In practice networks that use Batch Normalization are significantly more robust to bad initialization. Additionally, batch normalization can be interpreted as doing preprocessing at every layer of the network, but integrated into the network itself in a differentiable manner.

Batch normalization can implicitly regularize the model and in many cases, it is preferred over Dropout. Currently, the most widely accepted explanation of BatchNorm's success, as well as its original motivation, relates to the so-called internal covariate shift (ICS). Informally, ICS refers to the change in the distribution of layer inputs caused by updates to the preceding layers. It is assumed that such continual change negatively impacts training. The goal of BatchNorm was to reduce ICS and thus remedy this effect.

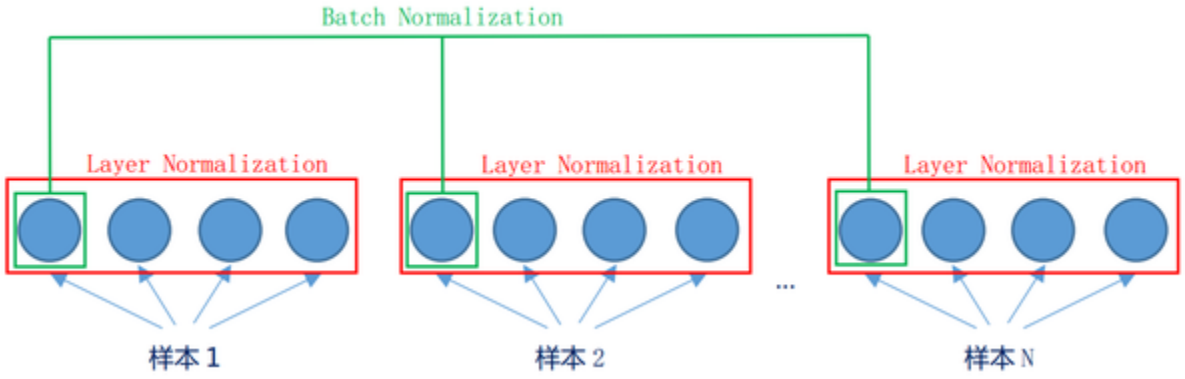
During the evaluation, this running mean/variance (from the training) is used for normalization.

Tricks when using BN:

- Probably Use Before the Activation
- Use Large Learning Rates
- Less Sensitive to Weight Initialization
- Don't Use With Dropout - The reason is that the statistics used to normalize the activations of the prior layer may become noisy given the random dropping out of nodes during the dropout procedure.
-

5) Layer Normalization

Layer normalization normalizes input across the features instead of normalizing input features across the batch dimension in batch normalization. The authors of the paper claim that layer normalization performs better than batch norm in the case of RNNs and transformers (In NLP tasks, the sentence length often varies -- thus if using batchnorm, it would be uncertain what would be the appropriate normalization constant (the total number of elements to divide by during normalization) to use. Different batches would have different normalization constants which leads to instability during the course of training)

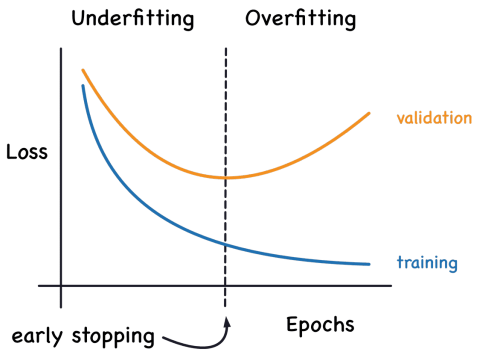


Batch Normalization and Layer Normalization are performed in different "directions". As presented in the picture, for batch normalization, input values of the same neuron (of the same channel) from different images in one mini-batch are normalized. In layer normalization, input values for different neurons (different channels) in the same layer are normalized without consideration of a mini-batch.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

6) Early stopping

It refers to the process of stopping the training when the training error is no longer decreasing but the validation error is starting to rise.

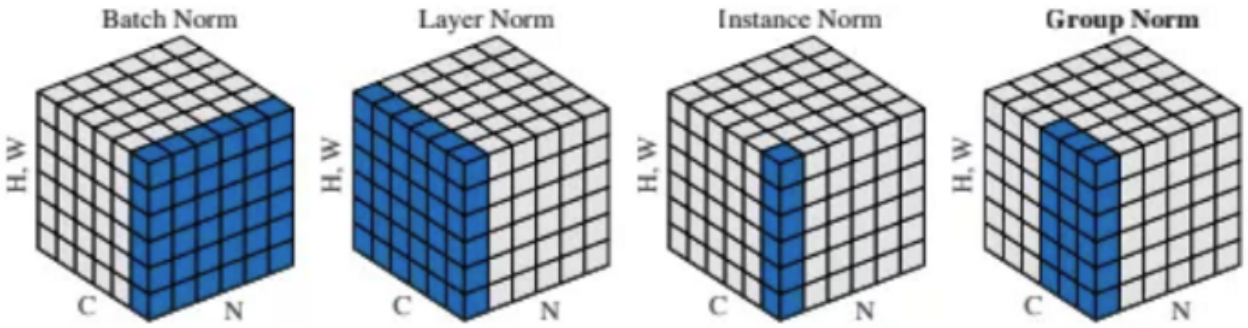


7) Instance Normalization

Instance normalization is very similar to layer normalization in the sense that they are both applied on each training sample independently.

- When to use it? use instance normalization for image classification where the class label should not depend on the contrast of the input image (since we normalize across the spatial locations of a single image, we normalize the contrast). In addition, when changing the style (like style transfer), or the batch size is very small for BN.

The difference between LN and IN is that for each training sample, IN normalizes across each channel separately -meaning across spatial locations only (see image below):



A visual comparison of various normalization methods

batch normalization - normalizes across the batch and spatial locations

instance normalization - normalized across the spatial locations of each training image

layer normalization - normalizes across the spatial locations and the channels of each training image

8) Parameter sharing

Instead of penalizing model parameters, it forces a group of parameters to be equal. This can be seen as a way to apply our previous domain knowledge to the training process. Various approaches have been proposed over the years but the most popular one is by far Convolutional Neural Networks. Convolutional Neural Networks take advantage of the spatial structure of images by sharing parameters across different locations in the input. Since each kernel is convoluted with different blocks of the input image, the weight is shared among the blocks instead of having separate ones.

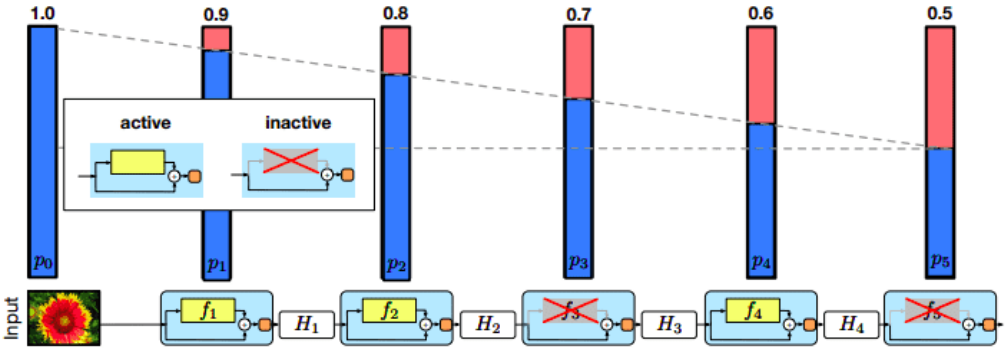
9) Stochastic depth

Stochastic depth goes a step further. It drops entire network blocks while keeping the complete model during testing. The most popular application is in large ResNets where we bypass certain blocks through their skip connections.

$$Hl = \mathrm{ReLU}(bl\,fl(Hl-1) + \mathrm{id}(Hl-1))$$

where b_l is a Bernoulli random variable that shows if a block in layer l is active or inactive. if b_l =0 the block will be inactive and if b_l =1 active.

In particular, Stochastic depth drops out each layer in the network that has residual connections around it. It does so with a specified probability p that is a function of the layer depth. (for each layer, the probability is different for skipping its block).



Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

10) Data augmentations

Data augmentation refers to the process of generating new training examples for our dataset. More training data means lower model variance, a.k.a lower generalization error. Simple as that. It can also be seen as a form of noise injection in the training dataset.

1) Basic Data Manipulations.

The first simple thing to do is to perform geometric transformations on data. Most notably, if we're talking about images we have solutions such as: flipping, cropping, rotations, translations, sharpness, contrast, solarize, equalize, color, brightness, etc.
CutOut (= Random Erasing) is a commonly used idea where we remove certain image regions. Another idea, called **MixUp**, is the process of blending two images from the dataset into one image. **CutMix** combines Cutout and Mixup - and is based on the issue of information loss and inefficiency present in regional dropout strategies. Instead of removing pixels and filling them with black or grey pixels or Gaussian noise, you replace the removed regions with a patch from another image, while the ground truth labels are mixed proportionally to the number of pixels of combined images.

2) Feature space augmentation

Another method for applying simple transformations such as adding noise, interpolating, or extrapolating between them. Instead of transforming data in the input space as above, we can apply transformations to the feature space. Example for producing these transformations - (1) An autoencoder learns a feature space from unlabeled data, representing each image using latent representation. (2) Applying transformation in feature space (3) once the new latent representations have been created, they can be decoded to generate a new image.

3) GAN-based Augmentation

Generative Adversarial Networks have been proven to work extremely well on data generation so they are a natural choice for data augmentation. (train a GAN to generate images from the dataset, and then use the generator to produce more images).

4) Mete-Learning.

In meta-learning, we use neural networks to optimize other neural networks by tuning their hyperparameters, improving their layout, and more. A similar approach can also be applied in data augmentation. In simple terms, we use a classification network to tune an augmentation network into generating better images. Example: We feed random images to an Augmentation Network (most likely a GAN), which will generate augmented images. Both the augmented image and the original are passed into a second network, which compares them and tells us how good the augmented image is. After repeating the process the augmentation network becomes better and better at producing new images.

11) Repeated Augmentation

In RA we form an image batch B by sampling [B/m] different images from the dataset, and transforming them up to m times by a set of data augmentations to fill the batch. The key difference with the standard sampling

scheme in SGD is that samples are not independent, as augmented versions of the same image are highly correlated (they include multiple data-augmented instances of the same image in one optimization batch).

12) RandAugment

RandAugment is a stochastic data augmentation routine for vision data and composed of strong augmentation transforms like color jitters, Gaussian blurs, saturations, etc. along with more traditional augmentation transforms such as random crops.

RandAugment has two parameters:

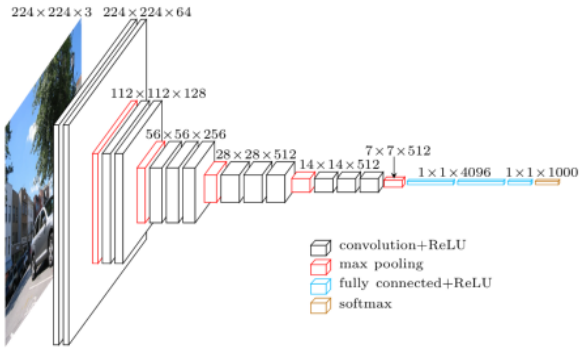
- n - denotes the number of randomly selected augmentation transforms to apply sequentially
- m - the strength of all the augmentation transform

13) Exponential Moving Average

Ordinarily, a moving average is a process of taking an average of the last few observations. So each time you get a new observation, you discard the earliest one and recompute the average. The number of observations you keep is called the "window". An exponential moving average is a weighted average over all of the observations, where the weights are an exponential series. The last observation is given some weight, say 90%. The observation before that gets 90% of the remaining 10% or 9%. Before that is 90% of the remaining 1% or 0.9%, and so on.

- It's usually used for time series data, very common for analyzing stock prices. When you are looking at real values data that varies with time, it is possible that there is noise in the signal (instead of changing smoothly it moves up and down a lot). You want to somehow still make sense of the general direction that the signal is moving, but simply looking at a specific point in time or a specific window might not tell you much about the trend.

Convolutional Neural Networks

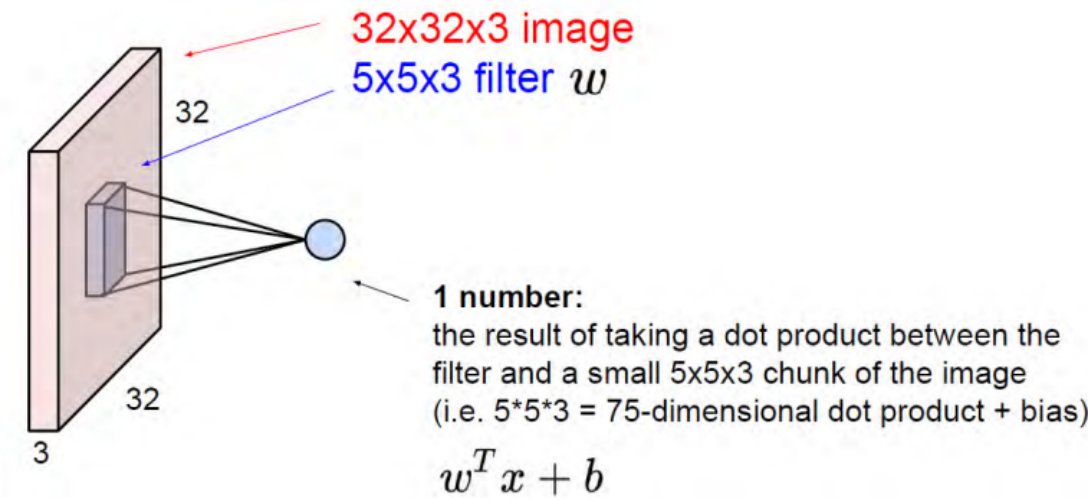


The layers are arranged in such a way so that they detect simpler patterns first (lines, curves, etc.) and more complex patterns (faces, objects, etc.) further along.

Convolution Layer

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

This layer performs a dot product between two matrices, where one matrix is the set of learnable parameters otherwise known as a kernel, and the other matrix is the restricted portion of the receptive field.



Computing the size of the feature map after convolution

If we have an input of size WxWxC, we can compute the spatial size of the output volume as a function of the input volume size (W), the receptive field size of the Conv Layer neurons (F), the stride with which they are applied (S), and the amount of zero padding used (P) on the border using the following formula:

$$W_{out} = \frac{W - F + 2P}{S} + 1$$

Implementation of convolution layer as matrix multiplication

say we have input of size (227,227,3) and we want to convolve it with a kernel of size (11,11,3) and stride 4 to produce 96 feature maps.

steps:

1) We want to partition the image into chunks, s.t each chunk will be multiplied with the kernel. Each chunk is of size 11x11x3=363.
   How many chunks like this do we have?
   Answer: the number of chunks is exactly the number of elements in each output map (that we have the formula to compute!). So the number of chunks is: (227-11)/4 + 1 = 55
   Thus, we have 55x55=3025 chunks like this that we need to multiply by the kernel.
   So, we can reshape the image to a matrix with the shape: (3025,363). It means that we have 3025 chunks, and each chunk is of size 363.
2) Now we reshape the kernel in a similar way - the number of elements in the kernel (the weight) is the same as the number of elements in each chunk. If we flatten the kernel we get indeed 363 elements. We want to output 96 feature maps, so we need to have 96 different kernels. Thus, the size of the kernel matrix becomes: (96,363).
3) Let's do a dot product between the both matrices!
   $X \cdot W^T -> (3025,\ 363) \times (363,\ 96) = (3025, 96)$
4) The result must finally be reshaped back its proper output dimension (55x55x96).

Number of parameters

input: 32*32*3. We want to convolve it with a kernel of size (5,5,3) to produce 10 activation maps. Remember that the number of channels in the filter needs to be the same size as the number of channels in the input since we place the 5*5*3 filter in the appropriate position (starting from the upper leftmost position) and then multiply all the 5*5*3 numbers from the filter with the corresponding numbers in the image and then sum all the results. The number of parameters are (weights) thus : for each activation map we have 5*5*3 +1 parameters(+1 for bias) . we want 10 activation maps as output thus we need 10 filters of size 5*5*3.
Thus in total: 10*(5*5*3+1)

1x1 convolution
A 1x1 convolution simply maps an input pixel with all its channels to an output pixel, not looking at anything around itself. It is often used to reduce the number of depth channels since it is often very slow to multiply volumes with extremely large depths

- Prefer a stack of small filter CONV to one large receptive field CONV layer - Intuitively, stacking CONV layers with tiny filters as opposed to having one CONV layer with big filters allows us to express more powerful features of the input, and with fewer parameters. As a practical disadvantage, we might need more memory to hold all the intermediate CONV layer results if we plan to do backpropagation.
- Usually the order of layers is: Conv -> BN -> ReLU -> MaxPool

Setting the number of layers and their sizes -

Neural Networks with more neurons can express more complicated functions. Overfitting occurs when a model with high capacity fits the noise in the data instead of the (assumed) underlying relationship.
It seems that smaller neural networks can be preferred if the data is not complex enough to prevent overfitting. However, this is incorrect - there are many other preferred ways to prevent overfitting in Neural Networks (such as L2 regularization, dropout, input noise). In practice, it is always better to use these methods to control overfitting instead of the number of neurons.
The subtle reason behind this is that smaller networks are harder to train with local methods such as Gradient Descent: It's clear that their loss functions have relatively few local minima, but it turns out that many of these minima are easier to converge to, and that they are bad (i.e. with high loss). Conversely, bigger neural networks contain significantly more local minima, but these minima turn out to be much better in terms of their actual loss. Since Neural Networks are non-convex, it is hard to study these properties mathematically. In practice, what you find is that if you train a small network the final loss can display a good amount of variance - in some cases, you get lucky and converge to a good place but in some cases, you get trapped in one of the bad minima. On the other hand, if you train a large network you'll start to find many different solutions, but the variance in the final achieved loss will be much smaller. In other words, all solutions are about equally as good and rely less on the luck of random initialization.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

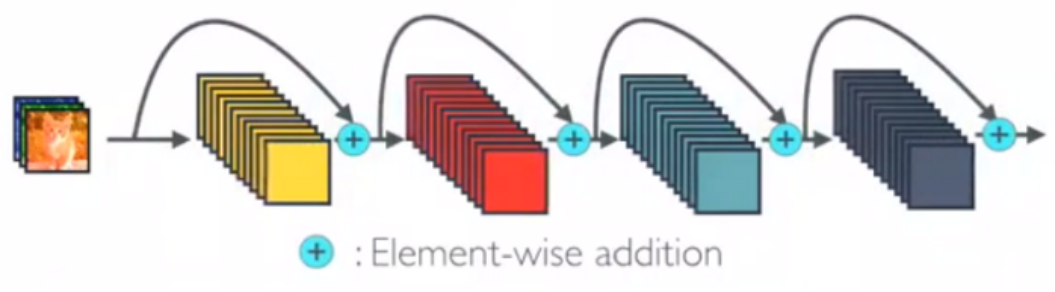
Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Convolution2D)	(None, 224, 224, 64)	1792
block1_conv2 (Convolution2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Convolution2D)	(None, 112, 112, 128)	73856
block2_conv2 (Convolution2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Convolution2D)	(None, 56, 56, 256)	295168
block3_conv2 (Convolution2D)	(None, 56, 56, 256)	590080
block3_conv3 (Convolution2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Convolution2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Convolution2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Convolution2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Convolution2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Convolution2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Convolution2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312
predictions (Dense)	(None, 1000)	4097000

Total params: 138357544

Several important CNNs

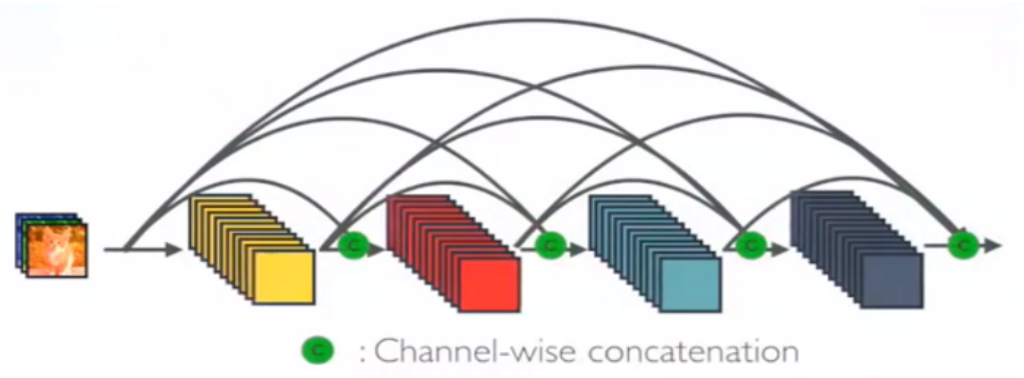
1) Inception network
2) **Reset** -
   In the beginning, researchers assume that “the deeper the better” when it comes to CNN. However, it has been noticed that after some depth, the performance degrades (this is because then the network is too deep, as the gradients keep flowing backward to the initial layers, this value keeps getting multiplied by each local gradient. Hence, the gradient becomes smaller and smaller, making the updates to the initial layers very small, increasing the training time considerably.) This was one of the bottlenecks of VGG. They couldn’t go as deep as wanted, because they started to lose generalization capability.
   However, we can give a counterexample that shows that deep networks should not be worse than shallow networks. Let us consider a shallower architecture and its deeper counterpart that adds more layers to it. There exists a solution by construction to the deeper model: the added layers are identity mapping, and the other layers are copied from the learned shallower model. The existence of this constructed solution indicates that a deeper model should produce no higher training error than its shallower counterpart.

How can we solve this problem? we need to make sure that the local gradient will be somehow at least 1. Which function always has derivative 1? the identity function!
Formally, denoting the desired underlying mapping as H(x), we let the stacked nonlinear layers fit another mapping of F(x) := H(x)−x. The original mapping is recast into F(x)+x. We hypothesize that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. To the extreme, if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers.

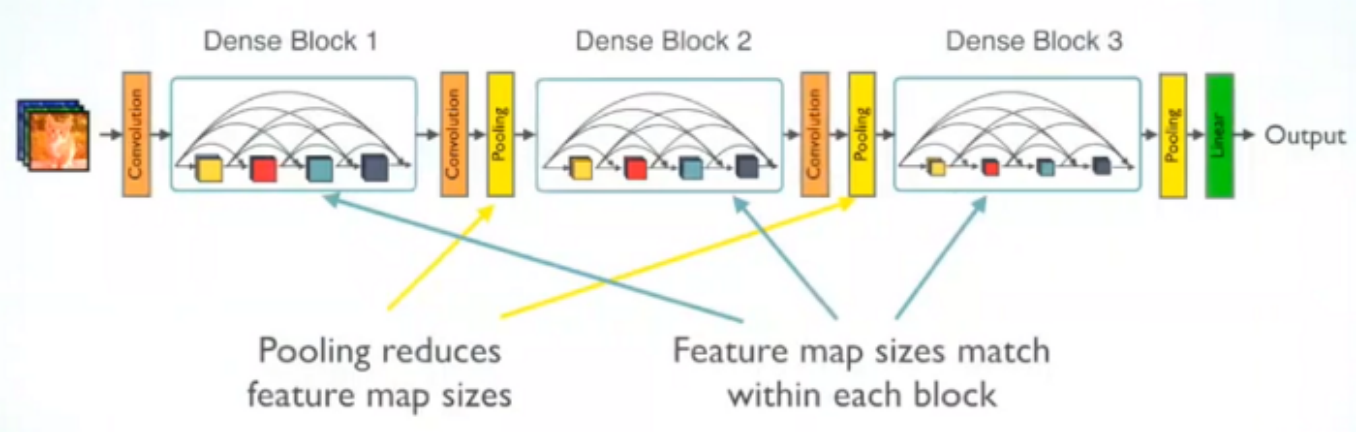


3) DenseNet

In DenseNet, each layer receives feature maps from all preceding layers and passes its own feature maps to all subsequent layers. Concatenation is used. Each layer is receiving a “collective knowledge” from all preceding layers.



1×1 Conv followed by 2×2 average pooling are used as the transition layers between two contiguous dense blocks. Feature map sizes are the same within the dense block so that they can be concatenated together easily. At the end of the last dense block, a global average pooling is performed and then a softmax classifier is attached.



Advantages of densenet:

1) strong gradient flow - The error signal can be easily propagated to earlier layers more directly. This is a kind of implicit deep supervision as earlier layers can get direct supervision from the final classification layer.
2) Parameter & Computational Efficiency
3) More Diversified Features
4) Maintains Low Complexity Features - In Standard ConvNet, the classifier uses the most complex features. In DenseNet, the classifier uses features of all complexity levels. It tends to give more smooth decision boundaries. It also explains why DenseNet performs well when training data is insufficient.

4) EfficientNet, EfficientNet-L2

In this paper, they study the process of scaling up ConvNets. In particular, investigate the central question: is there a principled method to scale up ConvNets that can achieve better accuracy and efficiency? Their empirical study shows that it is critical to balance all dimensions of network width/depth/resolution, and surprisingly such balance can be achieved by simply scaling each of them with constant ratio.

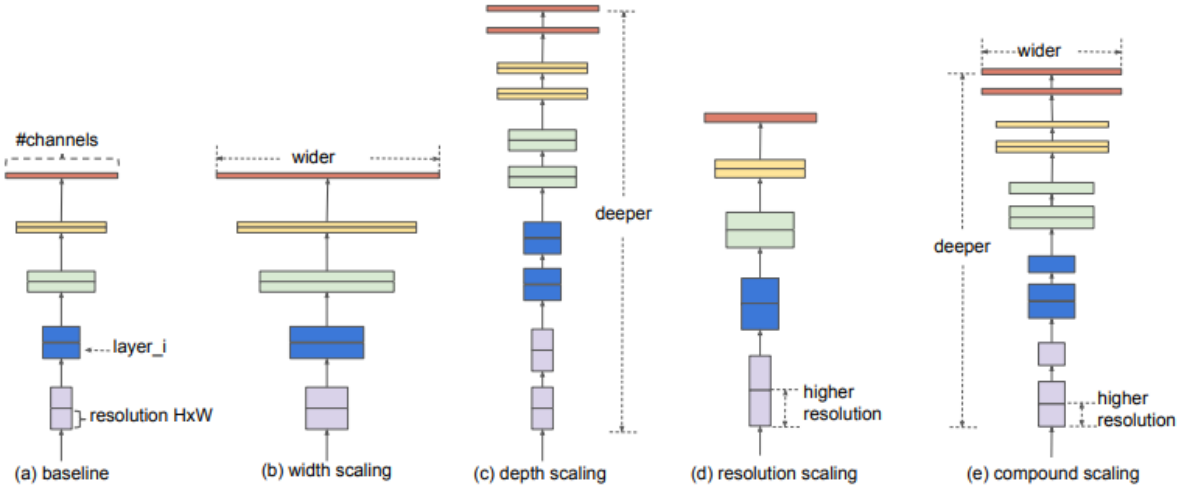
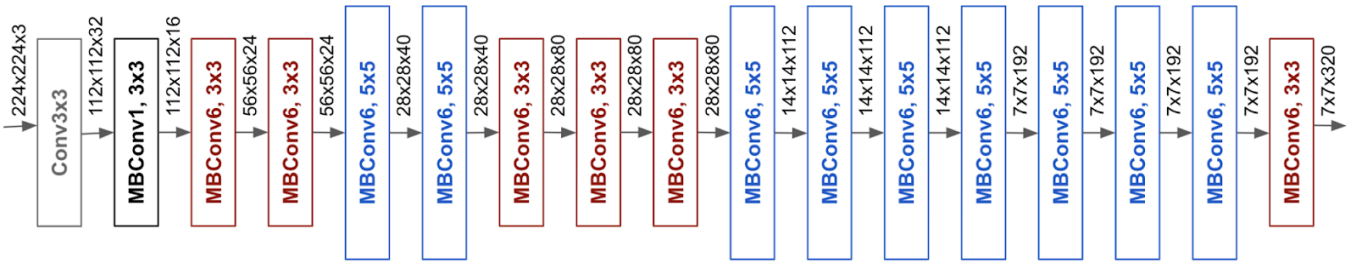


Figure 2. **Model Scaling.** (a) is a baseline network example; (b)-(d) are conventional scaling that only increases one dimension of network width, depth, or resolution. (e) is our proposed compound scaling method that uniformly scales all three dimensions with a fixed ratio.

- depth scaling - adding number of layers. intuition - deeper network can capture richer and more complex features.

- width scaling - increasing the number of channels. Wider networks tend to be able to capture more fine-grained features.
- resolution - Intuitively, we can say that in a high-resolution image, the features are more fine-grained and hence high-res images should work better.

The above three points lead to the observation: Scaling up any dimension of the network (width, depth, or resolution) improves accuracy, but the accuracy gain diminishes for bigger models.



combined scaling - The authors empirically observe that different scaling dimensions are not independent. Intuitively, for higher resolution images, we should increase network depth, such that the larger receptive fields can help capture similar features that include more pixels in bigger images. Correspondingly, we should also increase network width when resolution is higher, in order to capture more fine-grained patterns with more pixels in high-resolution images. These intuitions suggest that we need to coordinate and balance different scaling dimensions rather than conventional single-dimension scaling.

Global Average Pooling (GAP) instead of Fully Connected

In the last few years, experts have turned to global average pooling (GAP) layers to minimize overfitting by reducing the total number of parameters in the model. Similar to max-pooling layers, GAP layers are used to reduce the spatial dimensions of a three-dimensional tensor. However, GAP layers perform a more extreme type of dimensionality reduction, where a tensor with dimensions h×w×d is reduced in size to have dimensions 1×1×d. GAP layers reduce each h×w feature map to a single number by simply taking the average of all hw values. The ResNet-50 model added the GAP layer after the final Resnet block. The GAP layer is followed by one densely connected layer with a softmax activation function that yields the predicted object classes.

- **Object Localization**

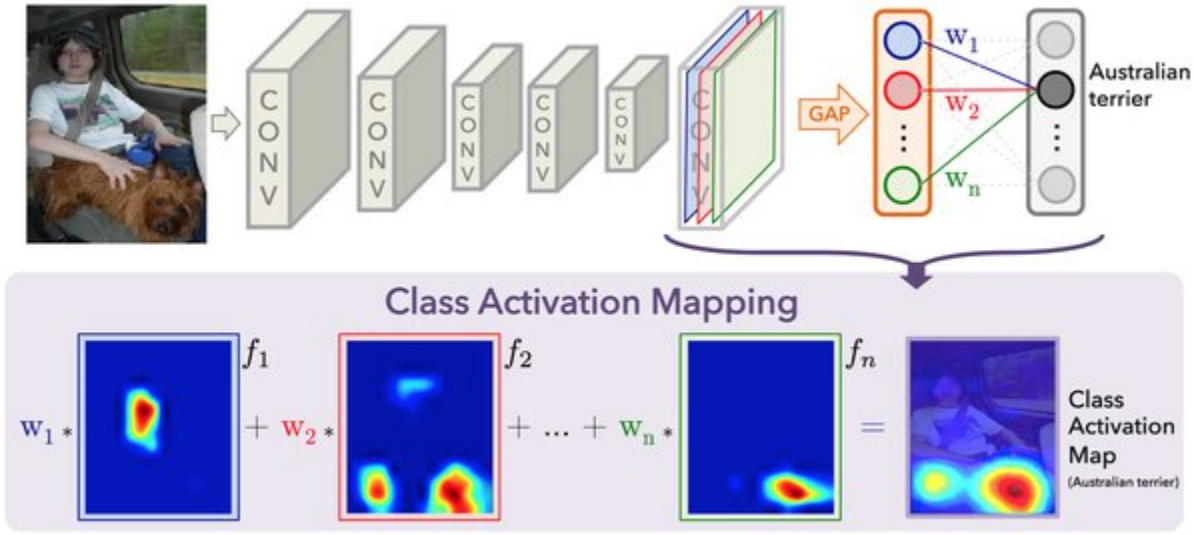
  "Learning Deep Features for Discriminative Localization" (2016) demonstrated CNNs with GAP layers (a.k.a. GAP-CNNs) that have been trained for a classification task can also be used for object localization. The main idea is that each of the activation maps in the final layer preceding the GAP layer acts as a detector for a different pattern in the image, localized in space. To get the class activation map corresponding to an image, we need only to transform these detected patterns to detected objects. This transformation is done by noticing each node in the GAP layer corresponds to a different activation map, and that the weights connecting the GAP layer to the final dense layer encode each activation map's contribution to the predicted object class. To obtain the class activation map, we sum the contributions of each of the detected patterns in the activation maps, where detected patterns that are more important to the predicted object class are given more weight.
  Example:

Let's say that after all the resnet blocks we get a tensor with shape: 7x7x2048. The AveragePooling2D GAP layer reduces the size of the preceding layer to (1,1,2048) by taking the average of each feature map. The next Flatten layer merely flattens the input, without resulting in any change to the information contained in the previous GAP layer.

The object category predicted by ResNet-50 corresponds to a single node in the final Dense layer (the argmax coordinate in the vector of size C where C is the number of classes) and, this single node is connected to every node in the preceding Flatten layer (where the flatten vector is of shape 2048). Let w_k represent the weight connecting the k-th node in the Flatten layer to the output node corresponding to the predicted image category.

Then, in order to obtain the class activation map, we need only compute the sum:

$$w_1 * f_1 + w_2 * f_2 + ... + w_{2048} * f_{2048}$$



Transfer learning

take a CNN, train on a large dataset (imagenet). Now we want to apply the features from this dataset to some small dataset that you care about. For the last fully connected layer that is going from the last features layer to the final scores, we need to reinitialize the matrix weight for this layer, freeze the weights of all the previous layers and train a linear classifier - only train the parameters of the last layer. If we have a bigger dataset we can finetune the whole network - as we have more and more data we should consider training more layers . We can also try to initialize the weights with the weights of training with Imagenet, and train the network with these initial weights, just with a lower learning rate: 1/10 of original LR.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

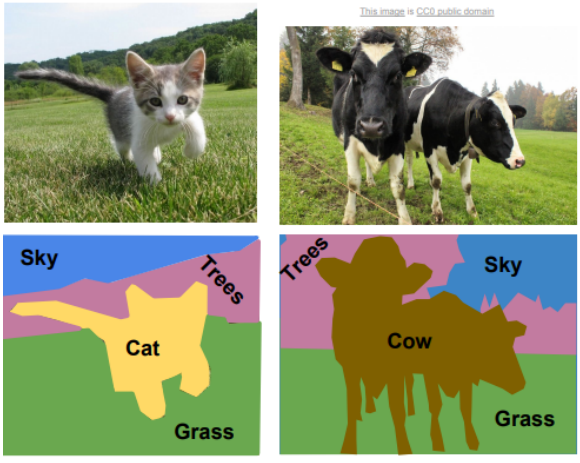
	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Examples for transfer learning models: VGG-16, Bert, GPT, Inception, encoder from Auto-encoder.

How to Update Neural Network Models With More Data

1. continue training the model on the new data only
2. continue training the model on the old and new data - We can update the model on a combination of both old and new data.
3. Ensemble of an existing model and new model fit on new data only - for example, train a new model only on the new data, and average the predictions from the existing model and the new model.
4. Ensemble Model With Model on Old and New Data - We can create an ensemble of the existing model and a new model fit on both the old and the new data.

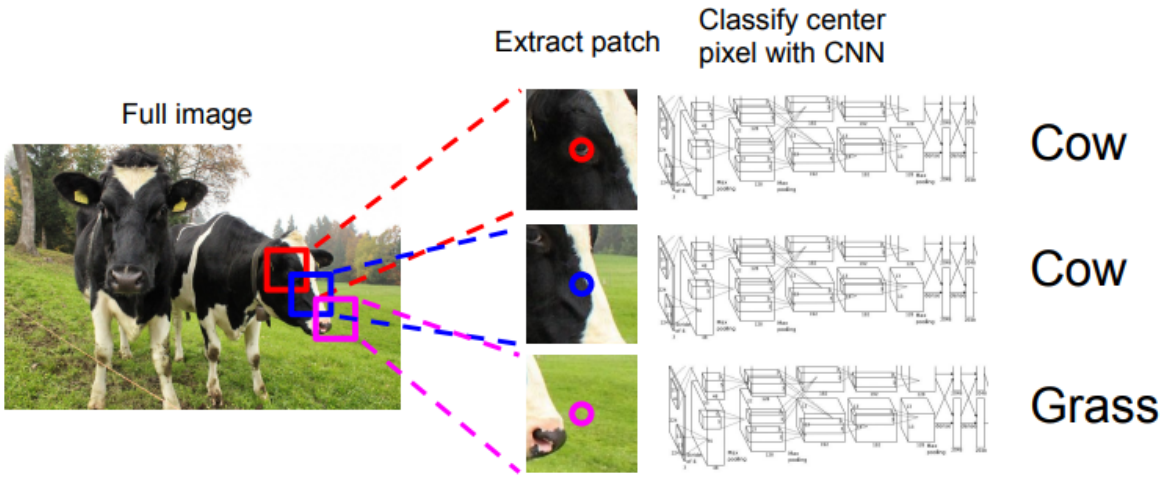
Semantic Segmentation



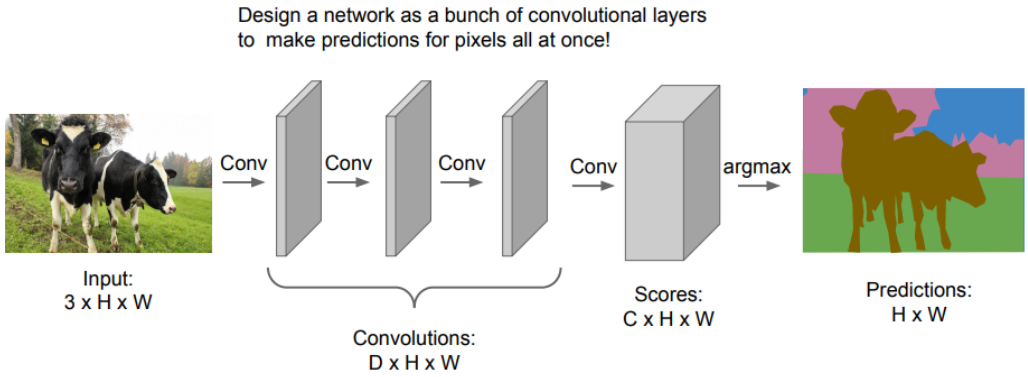
Goal: label each pixel in the image with a category label. Semantic segmentation doesn't differentiate instances (for example the 2 instances of the cow in the image above, are both “cow”). It only cares about pixels.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

First idea: sliding window. Take the input image and break it up into many small, tiny local crops of the image. Then, taking each of these crops and treating this as a classification problem: "for this crop, what is the category of the central pixel of the crop"? This is very inefficient because we need a separate crop for every pixel in the image (for each pixel, we will need a crop such that this pixel is the center of the crop). In addition, in this way we are not reusing shared features between overlapping patches.

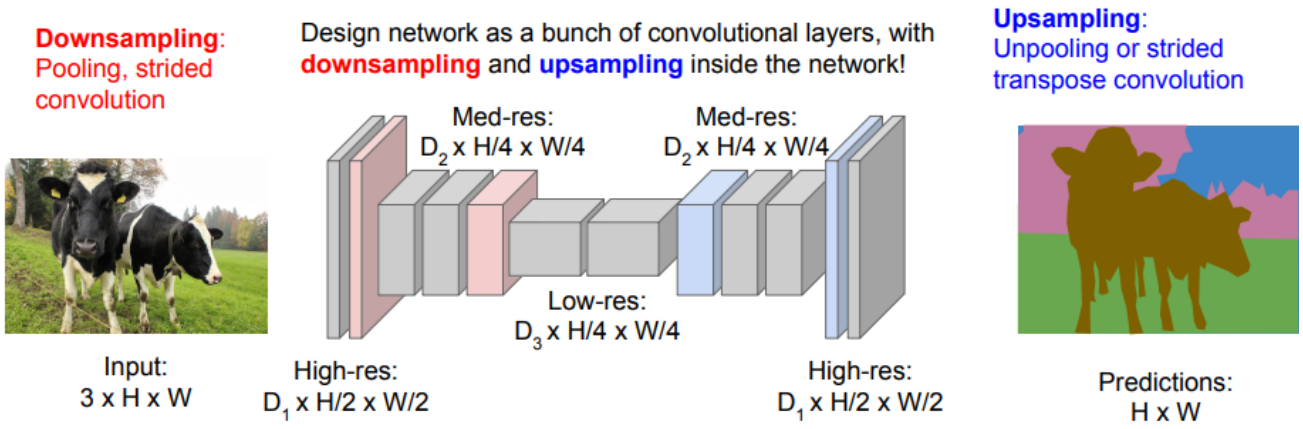


Second idea: fully convolutional - design a network as a bunch of convolutional layers to make predictions for pixels all at once. In this case, we use a fully convolutional network that takes an input of shape (3,H,W) and outputs at the end a feature map of shape (C, H, W) where C is the number of categories. This way, for each pixel we will have its category. We treat the final tensor of the network as just giving our classification scores for every pixel in the input image. We compute cross-entropy loss for every pixel in the output image: between every score in the output tensor and every ground truth category corresponding to this pixel. Then we take the sum/average over space. This is highly computationally expensive in space since all the convolutions are keeping the same size of the input image.



Third idea: instead we will do convolution with downsampling and upsampling inside the network. Since the feature map obtained at the output layer is down-sampled due to the set of convolutions performed, we would want to up-sample it using an interpolation technique. The downsampling -can be done using pooling or stride convolution (convolution with stride=2 for example). The upsampling can be done using unpooling or transpose convolution.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



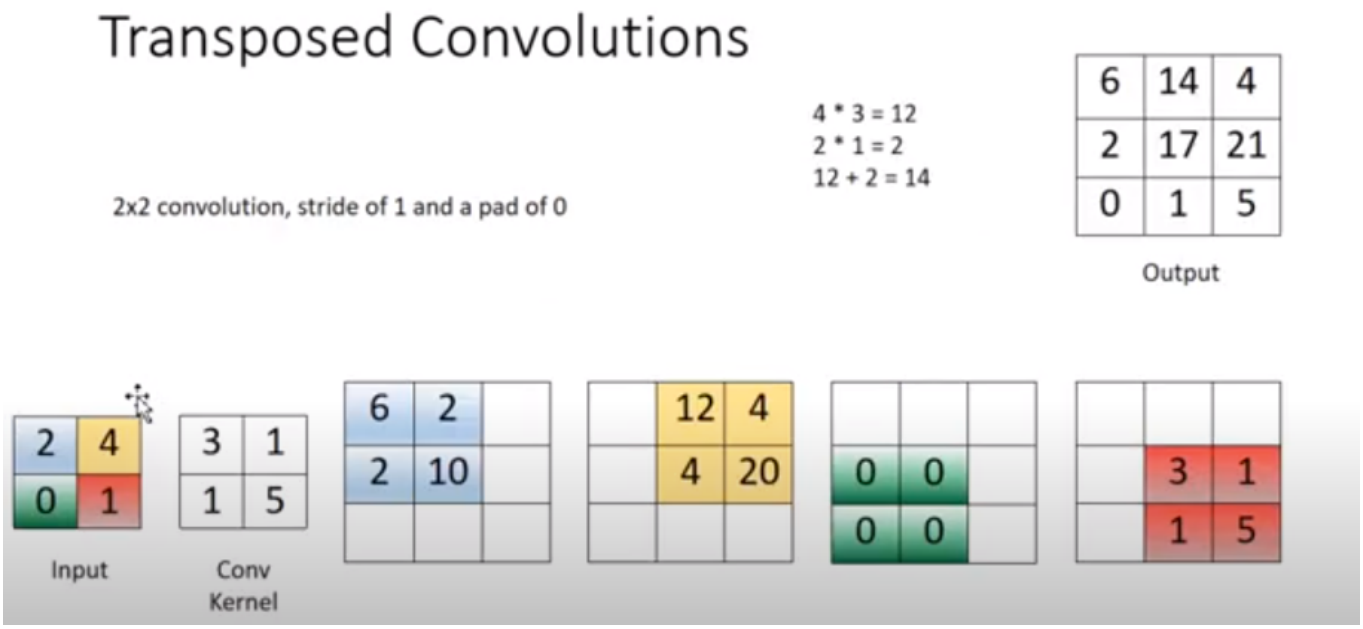
Transposed convolution

A transposed convolution layer is usually carried out for upsampling i.e. to generate an output feature map that has a spatial dimension greater than that of the input feature map. Just like the standard convolutional layer, the transposed convolutional layer is also defined by the padding and stride. These values of padding and stride are the one that hypothetically was carried out on the output to generate the input. i.e. if you take the output, and carry out a standard convolution with stride and padding defined, it will generate the spatial dimension same as that of the input.

take the value of the pixel in our input feature map and multiply the filter (say we have 3*3 filter with stride 2 and pad 1) by this scalar value. Then, copy the result to the corresponding region in the output (copy the 3*3 new values to the corresponding 3*3 region in the output). This scalar value is like a weight to the filter. it means that our output will be weighted copies of the filter.

example:

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



let's see how to compute: 

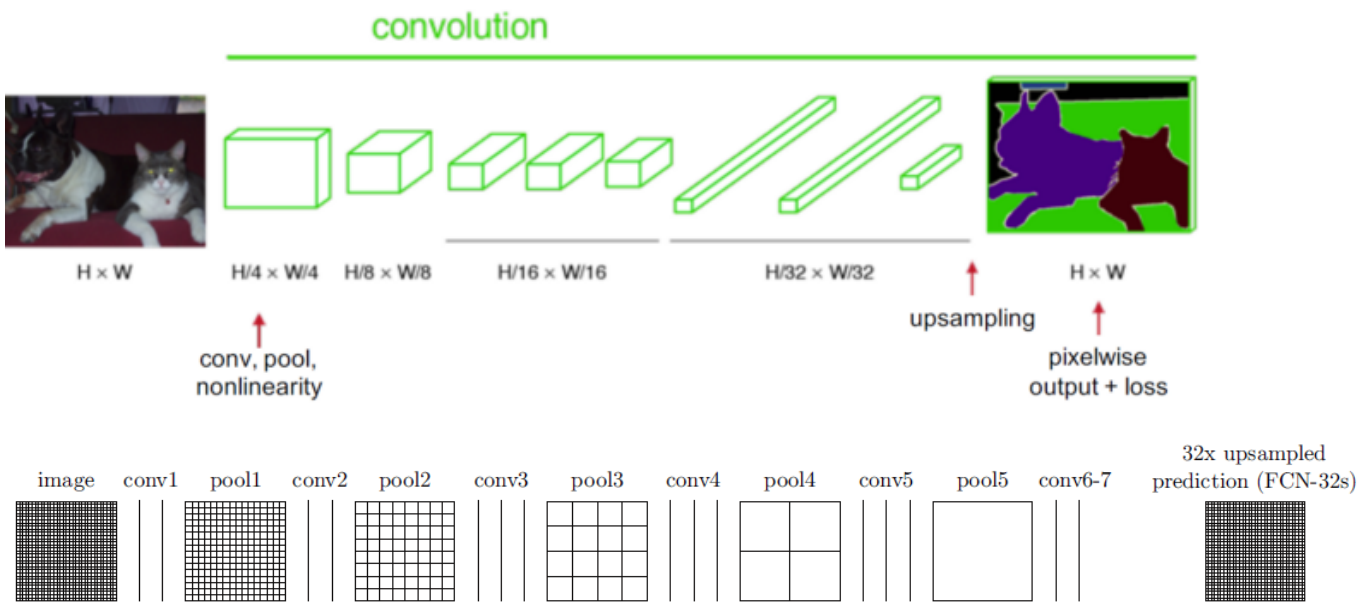
. we multiply the first value in the input (2) by the kernel: 2*3=6, thus the first element is 6. 2*1=2, thus the second element is 2. 2*1=2 again, and 1*5=10 and we finish. We continue to the second value in the input - “4”. Since the stride is 1, the 4x4 output values will be overlapping with the 2 values from the last calculation, and we will sum them at the end. Again, we multiply 4 by each element in the kernel. After this, we finish with the first row of the input and we continue to the second row.

Methods for image semantic segmentation

1) FCN (Fully Convolutional Network)

They build a fully convolutional network that consists only of convolution, pooling, and activation functions. While our reinterpretation of classification nets as fully convolutional yields output maps for inputs of any size, the output dimensions are typically reduced by subsampling. The classification nets subsample to keep filters small and computational requirements reasonable. This coarsens the output of a fully convolutional version of these nets, reducing it from the size of the input by a factor equal to the pixel stride of the receptive fields of the output units. . The problem is that the last conv layer produces coarse output maps, and we need to find a way to connect coarse outputs to dense pixels. One option is interpolation - For instance, simple bilinear interpolation computes each output y_ij from the nearest four inputs by a linear map that depends only on the relative positions of the input and output cells.

They took VGG16 and replaced that last layer with a fully convolutional layer.
First attempt (FCN-32s) - We append a 1 × 1 convolution with channel dimension 21 to predict scores for each of the PASCAL classes (including background) at each of the coarse output locations, followed by a deconvolution layer to bilinearly upsample the coarse outputs to pixel-dense outputs. However, The 32-pixel stride at the final prediction layer limits the scale of detail in the upsampled output.



After going through conv7 as above, the output size is small, then 32× upsampling is done to make the output have the same size as the input image. But it also makes the output label map rough. This is because deep features can be obtained when going deeper, spatial location information is also lost when going deeper. That means output from shallower layers has more location information. If we combine both, we can enhance the result. To combine, we fuse the output (by element-wise addition):

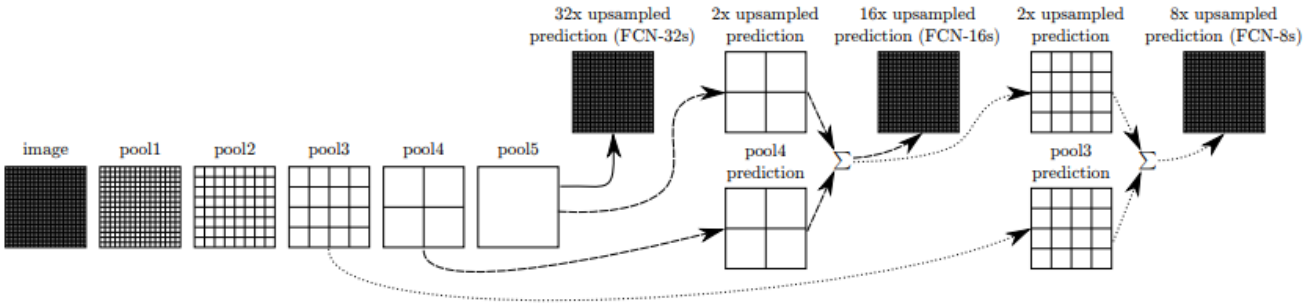
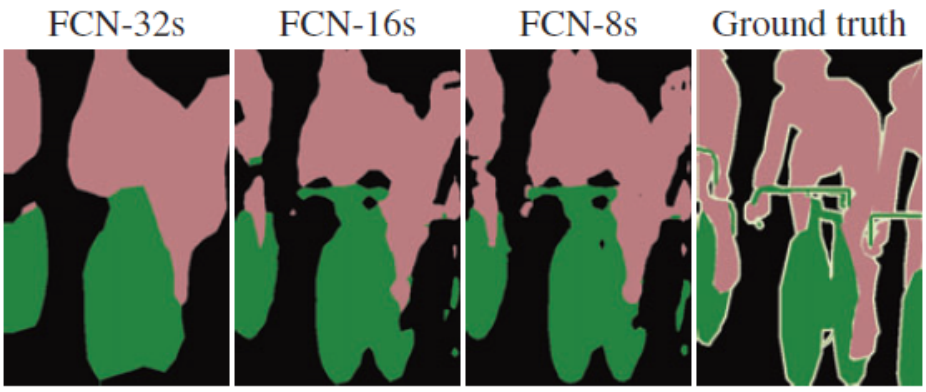


Figure 3. Our DAG nets learn to combine coarse, high layer information with fine, low layer information. Layers are shown as grids that reveal relative spatial coarseness. Only pooling and prediction layers are shown; intermediate convolution layers (including our converted fully connected layers) are omitted. Solid line (FCN-32s): Our single-stream net, described in Section 4.1, upsamples stride 32 predictions back to pixels in a single step. Dashed line (FCN-16s): Combining predictions from both the final layer and the `pool4` layer, at stride 16, lets our net predict finer details, while retaining high-level semantic information. Dotted line (FCN-8s): Additional predictions from `pool3`, at stride 8, provide further precision.

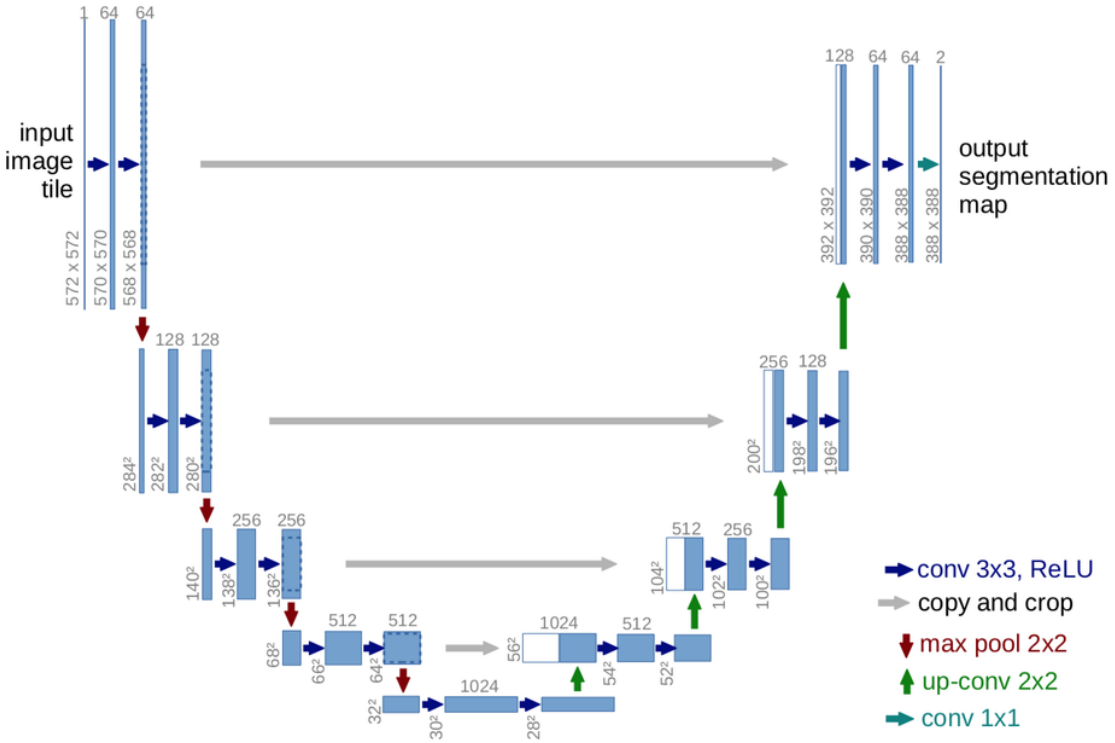
FCN-16s - We address this by adding links that combine the final prediction layer (The upsampled by x2 output from pool5) with lower layers with finer strides (pool4). Combining fine layers and coarse layers lets the model make local predictions that respect global structure. We add a 1 × 1 convolution layer on top of pool4 to produce additional class predictions. We fuse this output with the predictions computed on top of conv7 (convolutionalized fc7) at stride 32 by adding a 2× upsampling layer and summing both predictions. (See Figure 3). We initialize the 2× upsampling to bilinear interpolation but allow the parameters to be learned as described in Section 3.3. Finally, the stride 16 predictions are upsampled back to the image. We call this net FCN-16s.FCN-16s is learned end-to-end, initialized with the parameters

of the last, coarser net, which we now call FCN-32s. The new parameters acting on pool4 are zero-initialized so that the net starts with unmodified predictions

FCN-32s- We continue in this fashion by fusing predictions from pool3 with a 2× upsampling of predictions fused from pool4 and conv7, building the net FCN-8s.



2) Unet



U-net builds on top of the fully convolutional network from above.It also consists of an encoder which down-samples the input image to a feature map and the decoder which upsamples the feature map to

input image size using learned deconvolution layers. The main contribution of the U-Net architecture is the shortcut connections.
We saw above in FCN that since we down-sample an image as part of the encoder we lost a lot of information that can't be easily recovered in the encoder part. FCN tries to address this by taking information from pooling layers before the final feature layer.
U-Net proposes a new approach to solve this information loss problem. It proposes to send information to every upsampling layer in the decoder from the corresponding downsampling layer in the encoder as can be seen in the figure above thus capturing finer information whilst also keeping the computation low. Since the layers at the beginning of the encoder would have more information they would bolster the upsampling operation of the decoder by providing fine details corresponding to the input images thus improving the results a lot.

Training: the output of the network is K feature maps where K is the number of classes. so we have K vectors, each vector is the score vector for each pixel. We do softmax over the K options to get the max class for the pixel. So we denote by p_k(x) the probability that pixel x belongs to class k. we want to maximize the probability of the real class. So if we define Target(x) to be the true label of pixel x, we want to maximize log(p_target(x)(x)) -> maximize the probability that x belongs to class target(x).

overall the objective:

$$Loss = \sum_{x \in I} w(x) * log(P_{target(x)}(x)) \quad P_k(x) = \frac{e^{a_k(x)}}{\sum_{k'=1}^{K} e^{a_{k'}(x)}}$$

3) DeepLab

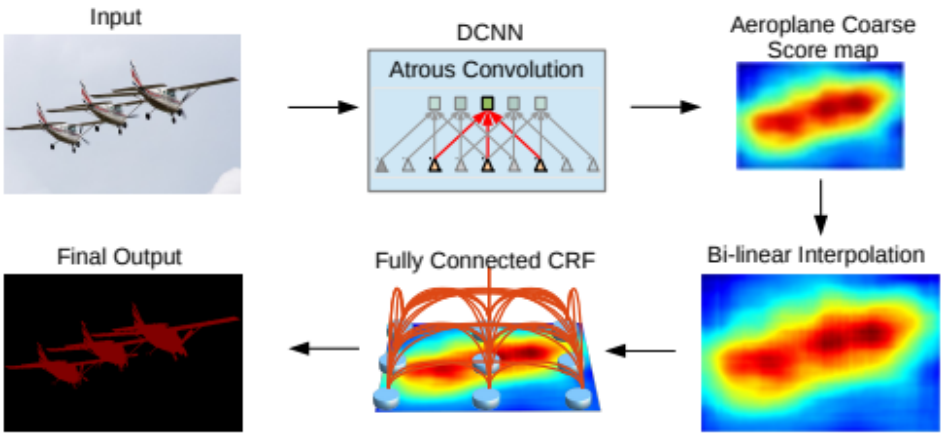


Fig. 1: Model Illustration. A Deep Convolutional Neural Network such as VGG-16 or ResNet-101 is employed in a fully convolutional fashion, using atrous convolution to reduce the degree of signal downsampling (from 32x down 8x). A bilinear interpolation stage enlarges the feature maps to the original image resolution. A fully connected CRF is then applied to refine the segmentation result and better capture the object boundaries.

They consider 3 challenges with applying deep CNN to semantic segmentation:

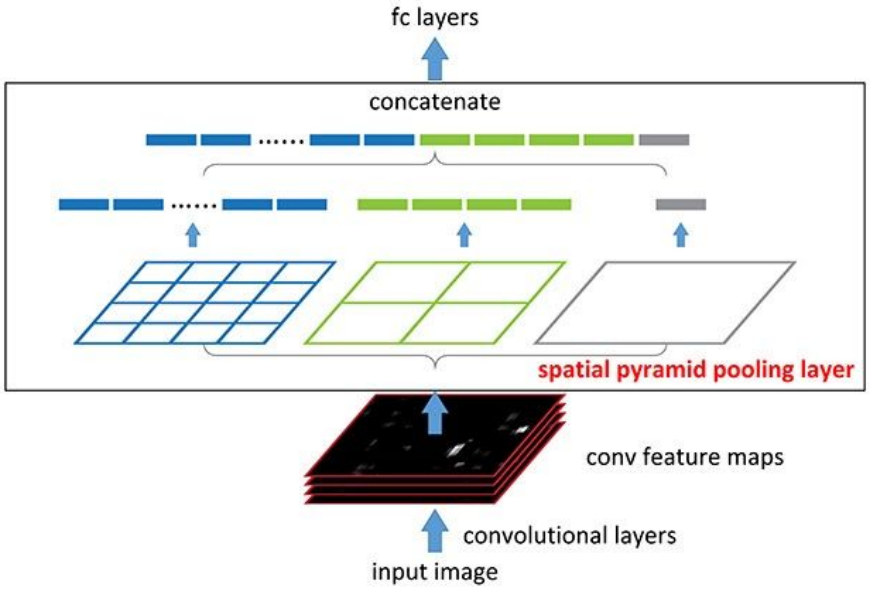
(1) Reduced feature resolution. This challenge is caused by the repeated combination of max-pooling and downsampling. This results in feature maps with significantly reduced spatial resolution. One solution is to use deconvolution layers to up-sample the resulting map. However, it requires additional memory and time.

Solution: Atrous convolution (dilated convolution). remove the downsampling operator from the last few max-pooling layers of DCNNs and instead **upsample the filters** in subsequent convolutional layers, resulting in increasing the receptive field (since we "enlarged" the kernel), thus getting a "smaller" image with less compute. Filter upsampling amounts to inserting holes ('trous' in French) between nonzero filter taps.

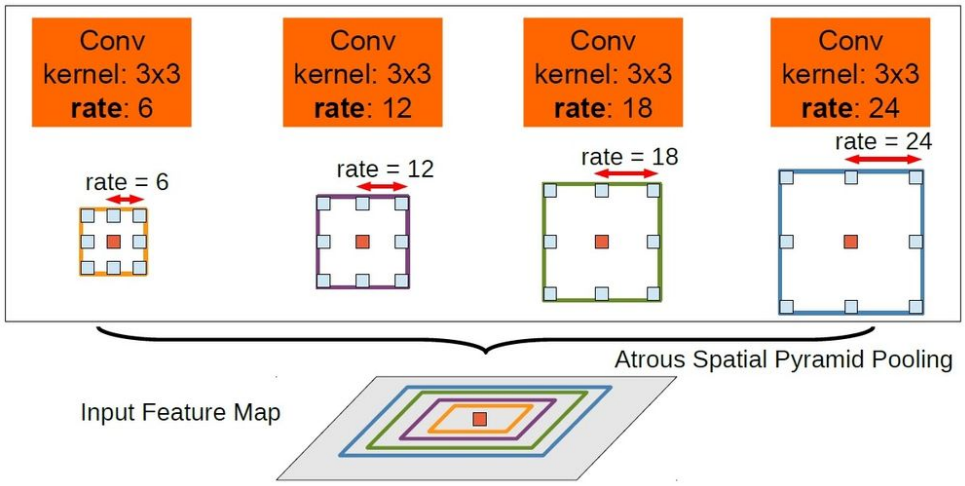
(2) Existence of objects at multiple scales. We want to be able to segment the object regardless of its size.

Solution: Atrous Spatial Pyramid Pooling (ASPP).

Spatial Pyramid Pooling is a concept introduced in SPPNet to capture multi-scale information from a feature map. With the SPP module, the network produces 3 outputs using three pooling operations, where one of them outputs only a single number for each map. The other one gives a 2 x 2 grid output for each map, and similarly, the last one gives a 4 x 4 output. This operation is applied to each feature map (256 maps in the above case) given out by the previous convolution operation. These outputs are flattened and then concatenated.



ASPP takes the concept of fusing information from different scales and applies it to Atrous convolutions. The input is convolved with different dilation rates and the outputs of these are fused together.



(3) reduced accuracy (especially in borders). All the fine details in the segmented object are lost. Case: Downsampling, max pooling. In classification, it is useful, because it is used to achieve invariance (we would like to be able to classify the object regardless to it's location, so we can do max pooling since we do not care about the location.

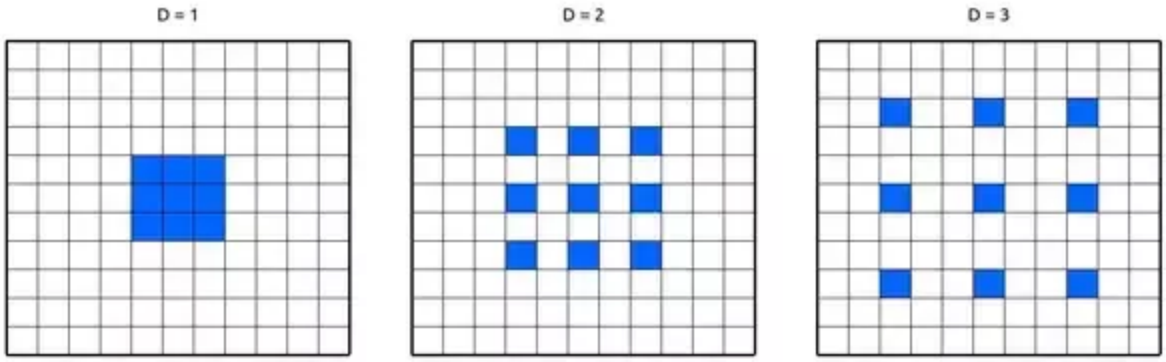
Solution: Conditional random fields (CRF). Conditional Random Field operates a post-processing step and tries to improve the results produced to define shaper boundaries. It works by classifying a pixel-based not only on it's label but also based on other pixel labels.

Dilated convolutions (astrous convolutions)

- bottom line: higher receptive field with less computation

Dilated convolution works by increasing the size of the filter by appending zeros(called holes) to fill the gap between parameters. The number of holes/zeroes filled in between the filter parameters is called by a term dilation rate. When the rate is equal to 1 it is nothing but the normal convolution. When the rate is equal to 2 one zero is inserted between every other parameter making the filter look like a 5x5 convolution. Now it has the capacity to get the context of 5x5 convolution while having 3x3 convolution parameters. Similarly for rate 3 the receptive field goes to 7x7.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



- Deeplab-v3+ suggested having a decoder instead of plain bilinear upsampling. The decoder takes a hint from the encoder layers to improve the results. The encoder output is upsampled 4x using bilinear upsampling and concatenated with the features from the encoder which is again upsampled 4x after performing a 3x3 convolution.

4) Global Convolution Network

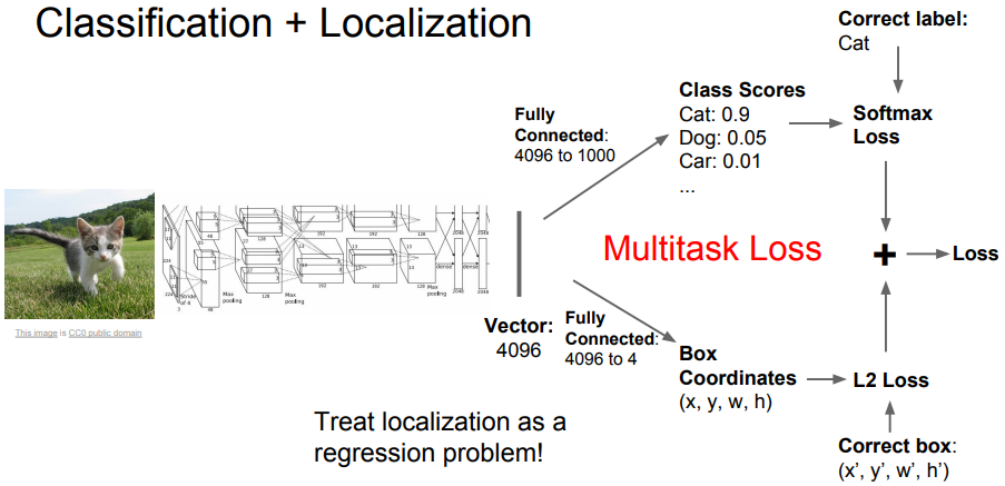
5) Spade

Classification + Localization

- A simpler version of Object Detection!

Goal: We want to assign a label category to the input image as in classification. However, we want additionally to predict the position of the object in the image - we want to draw a bounding box around the region of the object in the image. we assume that there is a single object in the image that we want to bound.

Key idea: We feed the deep convolutional neural network with an input image that outputs the final vector summarizing the content of the image.
Then we will have two fully connected layers: (1) FC layer that goes from the final vector to our class scores (2) FC layer that goes from that vector to four numbers where the four numbers are the height, width, and the x and y positions of the bounding box. we will have two losses: softmax loss and L2 loss between the bounding boxes.



Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

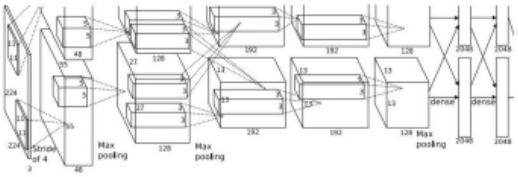
We can take a pretrained network on imagenet to the network before FC and then fine-tune the network.

- **Human Pose estimation**
  We are given as input a picture of a person and we want to output the positions of the joints for that person. This will allow us to predict what is the pose of the human (where are his arms, where are his legs, etc.)
  Simplifying assumption: Most of the people have the same number of joints.
  We will represent the pose as a set of 15 joint positions. For example, left/right foot, left/right knee, left/right shoulder, etc. Here, we will train a DNN that will output 14 coordinates (x,t) for each of those 14 joints.

Object detection

Goal: Locate the presence of objects with a bounding box and classes of the located objects in an image. This is different from classification+localization since there might be varying numbers of outputs for every input image
The problem here is that each image needs a different number of outputs since there are different number of objects in every image. Thus, we can't treat this problem as regression (as opposed to classification + localization).
Background:

(1) **Sliding Windows** - Apply a CNN to many different crops of the image, CNN classifies each crop as object or background. We take each crop, feed it to the convolutional neural network and the network makes a classification decision based on that input crop. Then, in addition to the categories that we care about, we will add an additional category called background (=there is no object in this crop) and now our network can predict background in case it does not see any of the categories we care about.
Problem - how do we choose the crop? the objects can appear at any size, viewpoint, etc.. With this strategy, we need to apply CNN to a huge number of locations and scales, which very computationally expensive!

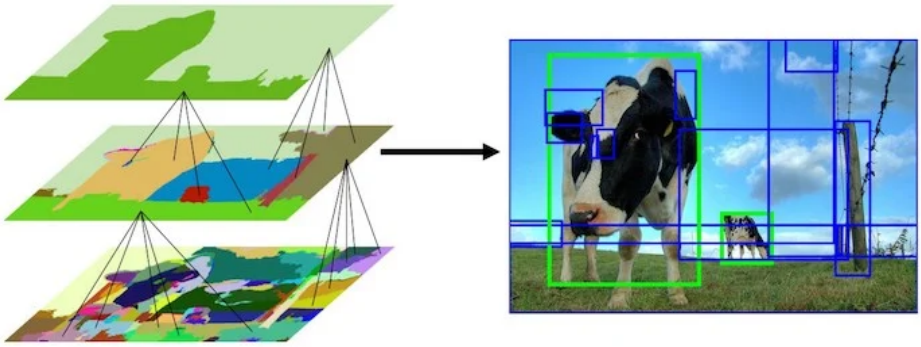


Dog? NO
Cat? NO
Background? YES

(2) **Region Proposals** -
find "blobby" image regions that are likely to contain objects (blobby region = regions that differ in properties, such as brightness or color, compared to surrounding regions. group of pixel values that forms a somewhat colony or a large object that is distinguishable from its background). It will give us like a thousand boxes where the object might be present.
For example **selective search algorithm**.

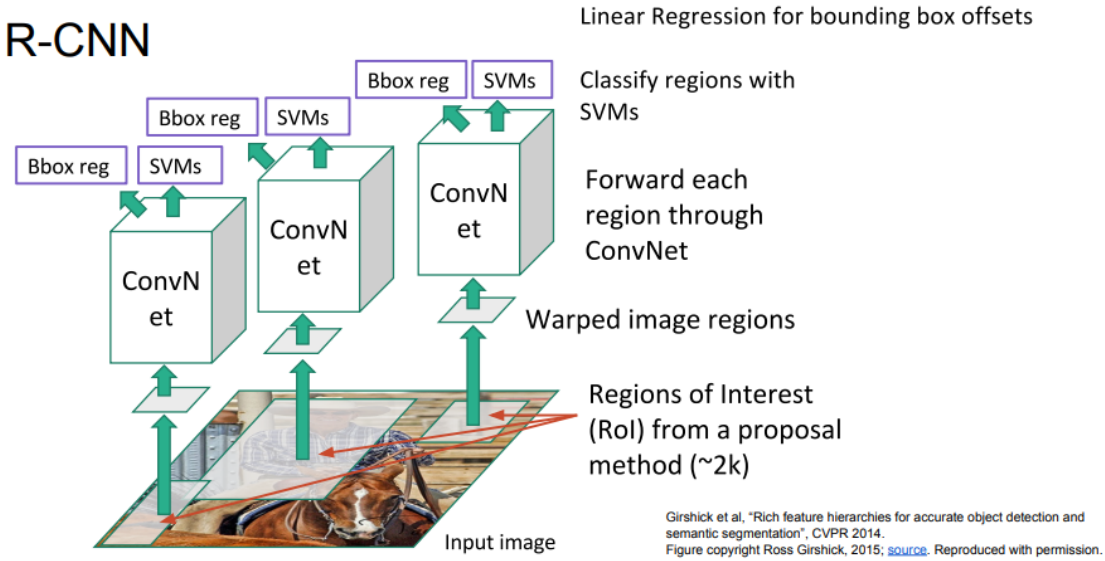
1) Selective Search starts by over-segmenting the image based on the intensity of the pixels using a graph-based segmentation method by Felzenszwalb and Huttenlocher.
2) Selective Search algorithm takes these oversegments as initial input and performs the following steps: a) Add all bounding boxes corresponding to segmented parts to the list of regional proposals b) Group adjacent segments based on similarity - color similarity, texture similarity, size

similarity, shape similarity, meta similarity measure. c) go to step a).
At each iteration, larger segments are formed and added to the list of region proposals. Hence we create region proposals from smaller segments to larger segments in a bottom-up approach.



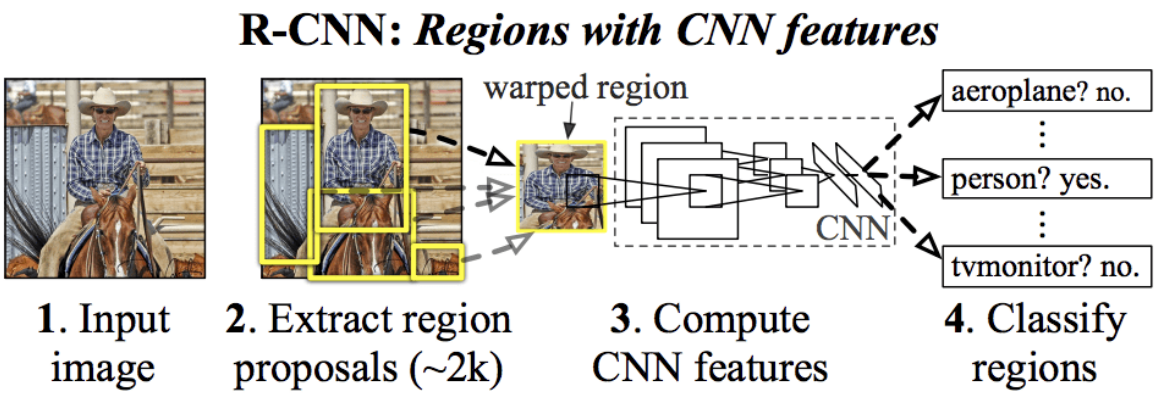
R-CNN

Given the input image, we run a regional proposal network. The regions proposals can be from different sizes, so we need to warp them to fixed square size. After we have all the region proposals, apply ConvNet to each of the proposed regions. Then, classify regions with SVM. In addition to the category labels for each of these proposals, it will also predict four numbers that are a kind of correction to the boxes predicted by the region proposal network.



Girshick et al, "Rich feature hierarchies for accurate object detection and semantic segmentation", CVPR 2014.
Figure copyright Ross Girshick, 2015; source. Reproduced with permission.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



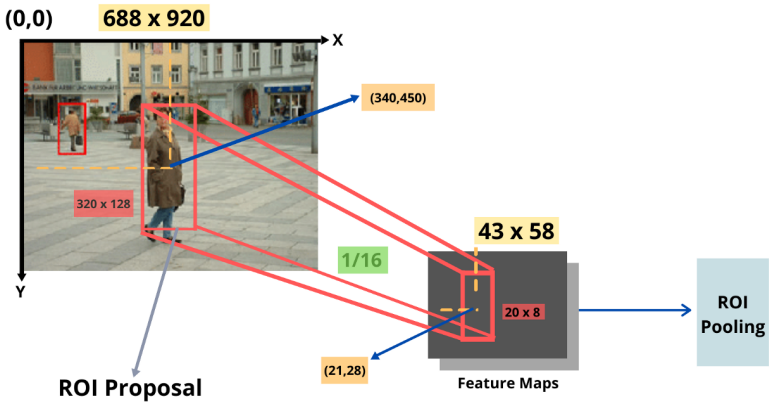
Fast R-CNN

The paper opens with a review of the limitations of R-CNN, which can be summarized as follows:

- **Training is a multi-stage pipeline**. Involves the preparation and operation of three separate models
- **Training is expensive in space and time**. Training a deep CNN on so many region proposals per image is very slow.
- **Object detection is slow**. Making predictions using a deep CNN on so many region proposals is very slow.

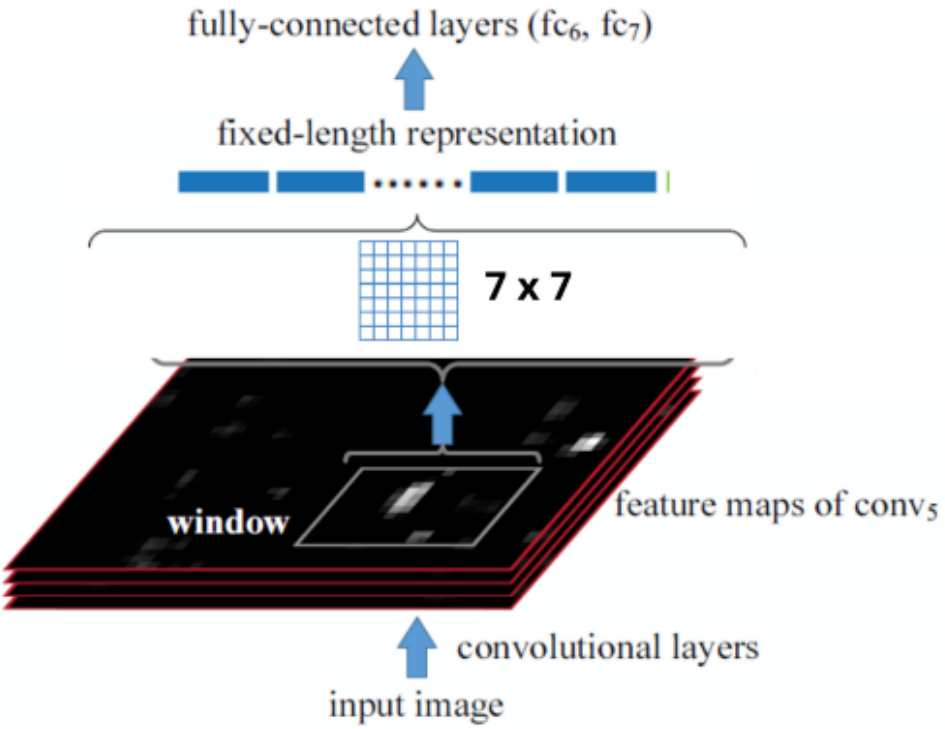
Pipeline:

1) The image is fed to underlying CNN just once and the selective search is ron on the other hand as usual.
2) The region proposals generated by the selective search are then projected onto the feature maps generated by the CNN. This process is called **ROI projection**. The idea of ROI projection is that we get the coordinates of the bounding box from the ROI proposal and we need to project them onto the feature maps by projecting the ROI proposal with respect to the subsampling ratio.
   Consider an image of size 688 x 920 is fed to a CNN whose subsampling ratio is 1/16. The resulting feature map's size then leads to 43 x 58 (688/16 x 920/16). Similarly, the size of the ROI Proposal 320 x 128, after subsampling leads to 20 x 8. Generally, the coordinates of bounding boxes are represented with Coordinates of mid-point of the box (X, Y), Width, Height. [X, Y, W, H]. Here, we consider the first notation. From the diagram, the mid-point of the ROI Proposal is (340,450) which leads to (21,28) in the feature map. In this way, the ROI Proposal is projected onto the feature map.



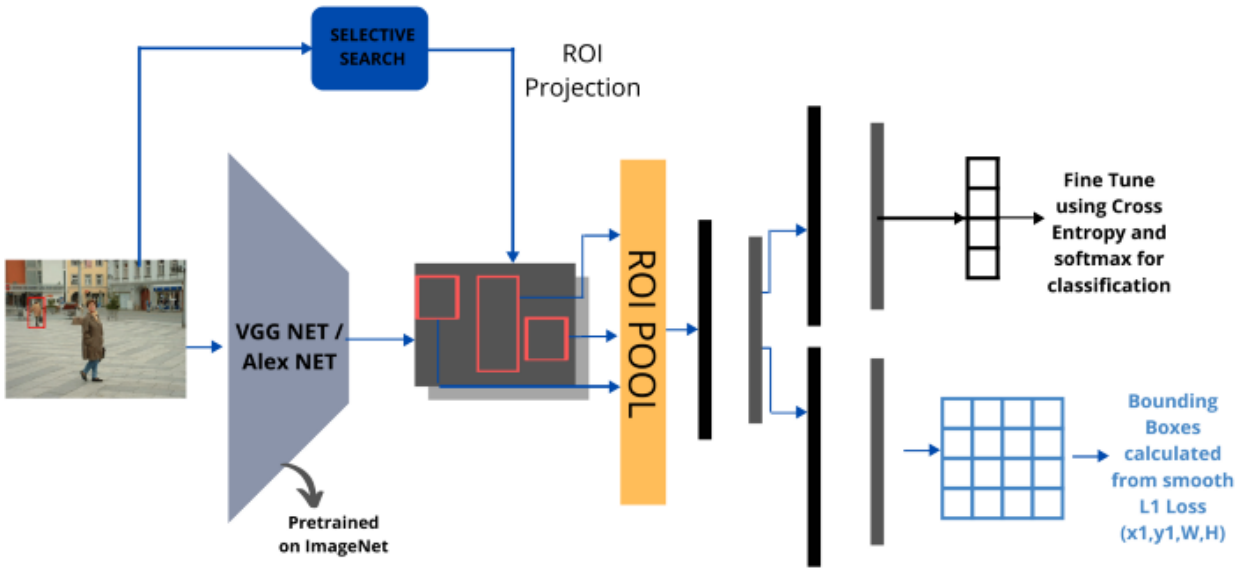
after we projected the region proposals to the feature maps, we need to wrap them to a fixed size using **ROI pooling.** Assume that the ROI bounding box after projecting it to the feature map is of size 14x14 and we want all the bounding boxes to have size 7x7. So they take they divide the height and the width of the bounding box with the size of the target bounding box: 14/7, 14/7 = (2,2) so we get a grid of 7x7 sub boxes, each box of size 2x2. Then each block is max pooled (we take the max value from each sub box) and the output is calculated.

3) note that there are several feature maps extracted from the CNN, so we need to pear this process for every feature map (projecting and pooling).

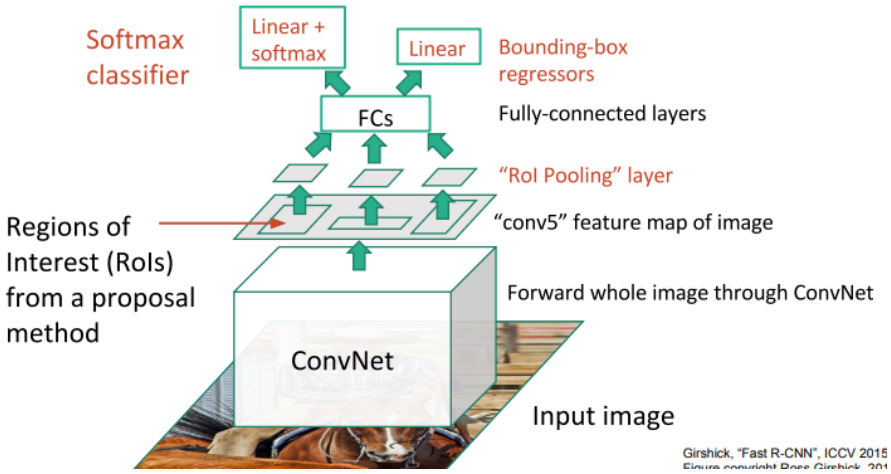


4) Once we have these warped crops from the convolutional feature map, we can run these through fully connected layers and predict the classification scores and linear regression offsets to the bounding boxes.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



The reason "Fast R-CNN" is faster than R-CNN is because you don't have to feed 2000 region proposals to the convolutional neural network every time. Instead, the convolution operation is done only once per image and a feature map is generated from it.



Faster R-CNN

Both of the above algorithms (R-CNN & Fast R-CNN) use selective search to find out the region proposals. Selective search is a slow and time-consuming process affecting the performance of the network. Faster R-CNN eliminates the selective search algorithm and lets the network learn the region proposals.

Observation: The convolution feature maps used by fast RCNN can also be used for generating region proposals. On top of these convolutional features, we construct an RPN by adding a few additional convolutional layers that simultaneously regress region bounds and objectness scores at each location on a regular grid.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Similar to Fast R-CNN, the image is provided as an input to a convolutional network which provides a convolutional feature map. Instead of using a selective search algorithm on the feature map to identify the region proposals, a separate network is used to predict the region proposals - **Region Proposal Network (RPN)**. The predicted region proposals are then reshaped using a RoI pooling layer which is then used to classify the image within the proposed region and predict the offset values for the bounding boxes.

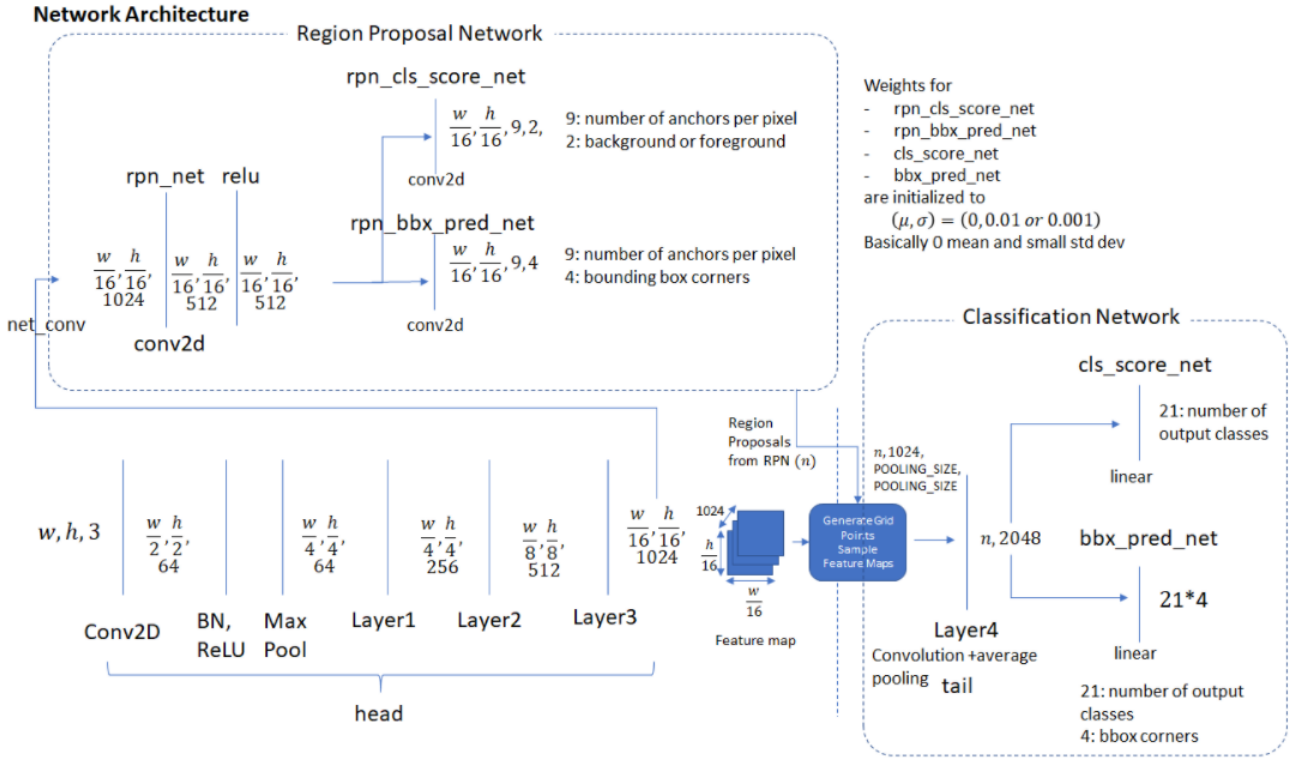
To unify RPNs with Fast R-CNN, we propose a training scheme that alternates between fine-tuning for the region proposal task and then fine-tuning for object detection, while keeping the proposals fixed.

Model

Faster R-CNN, is composed of two modules:

1) The first module is a deep fully convolutional network that proposes regions (RPN). The RPN module tells the Fast R-CNN module **where** to look.
2) The second module is the Fast R-CNN detector that uses the proposed regions. T

The entire system is a single, unified network for object detection.



Region proposal network (RPN)

Input: image of any size

output: A set of rectangles object proposals, each with an objectness score (if it is an object or background).

A fully convolutional network that generates proposals with various scales and aspect ratios. They introduce novel “anchor” boxes that serve as references at multiple scales and aspect ratios.

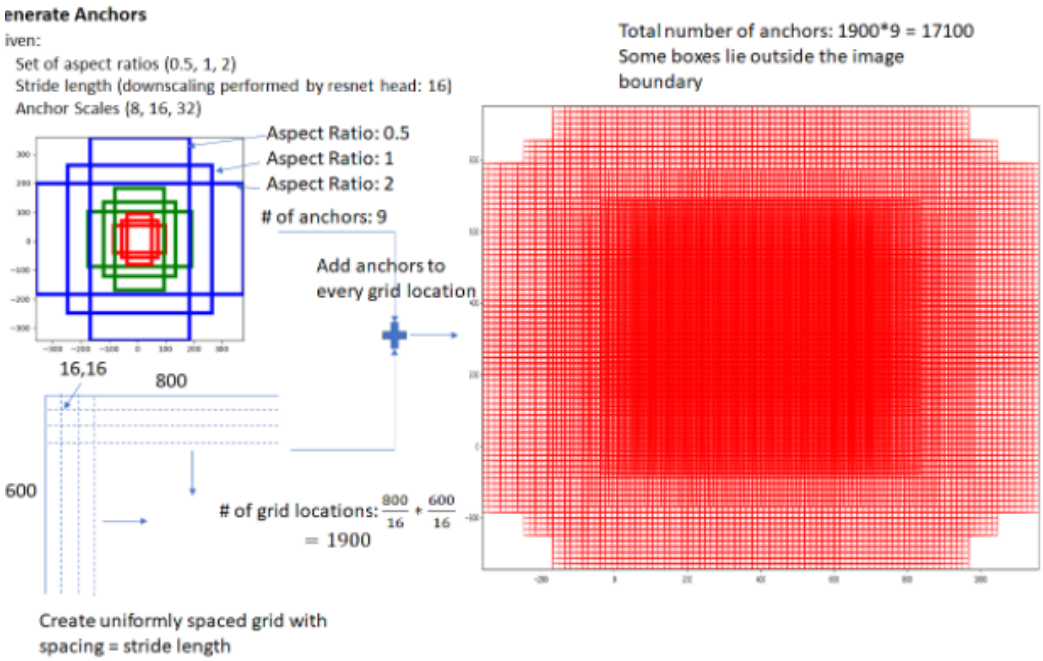
FULL architecture

a) feed the image to the head network to obtain feature maps

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

b) RPN:

1) first, we generate a fixed number of anchors by first generating 9 anchors of different scales and aspect ratios (just a list of anchors with base size 16*16 and different scales and ratios).



2) Region Proposal layer - From a list of anchors (from the previous stage), identify background and foreground anchors and then Modify the position, width and height of the anchors by applying a set of "regression coefficients" to improve the quality of the anchors (for example, make them fit the boundaries of objects better).
   - The region proposal layer runs feature maps produced by the head network through a convolutional layer. The output is passed through two 1x1-convolutional layers to produce background/foreground class scores and probabilities and corresponding bounding box regression coefficients. (for each convolution layer, the number of channels is 9*2 for the scores and 9*4 for the bounding boxes. These coefficients are from the regression to the GT boxes. We should update the boxes in the next stage according to these coefficients.
   - compute the ROIS - Now we want to update the anchors from 1) using the coefficients, and prune boxes based on their foreground score.
   - compute classification loss - cross-entropy between the ground truth scores and the predicted scores.
   - compute bounding box regression loss - L1 between the boxes predicted by the RPN and the target boxes.
   - return the ROIS (after updating with coefficients), and the losses

c) extract regions corresponding to these ROIs from the feature maps produced by the head network

d) These feature maps from c) are then passed through resnet layers followed by AVG pooling among the spatial dimension. To this end, we get n vectors (for n ROIs) each one with size 2048. These feature vectors are then passed through two fully connected layers. bbox_pred_net and cls_score_net. The cls_score_net layer produces the class scores for each bounding box (which can be converted into probabilities by applying softmax). The

bbox_pred_net layer produces the class-specific bounding box regression coefficients which are combined with the original bounding box coordinates produced by the proposal target layer to produce the final bounding boxes.

Note: It's good to recall the difference between the two sets of bounding box regression coefficients – one set produced by the RPN network and the second set produced by the classification network. The first set is used to train the RPN layer to produce good foreground bounding boxes (that fit more tightly around object boundaries). The second set of bounding box coefficients is generated by the classification layer. These coefficients are class-specific, i.e., one set of coefficients is generated per object class for each ROI box.

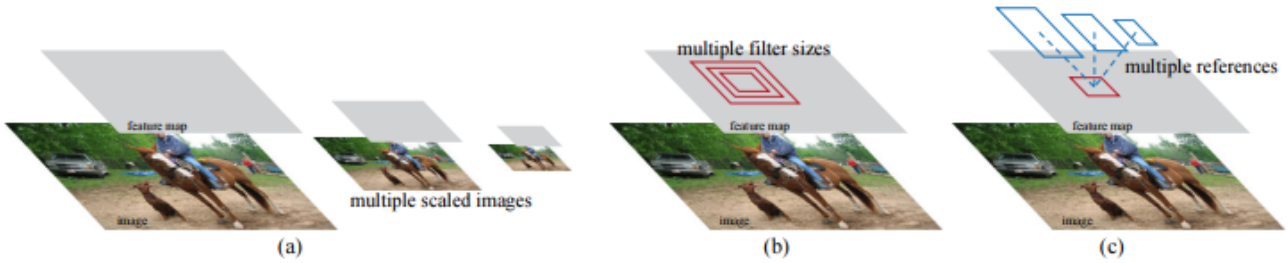
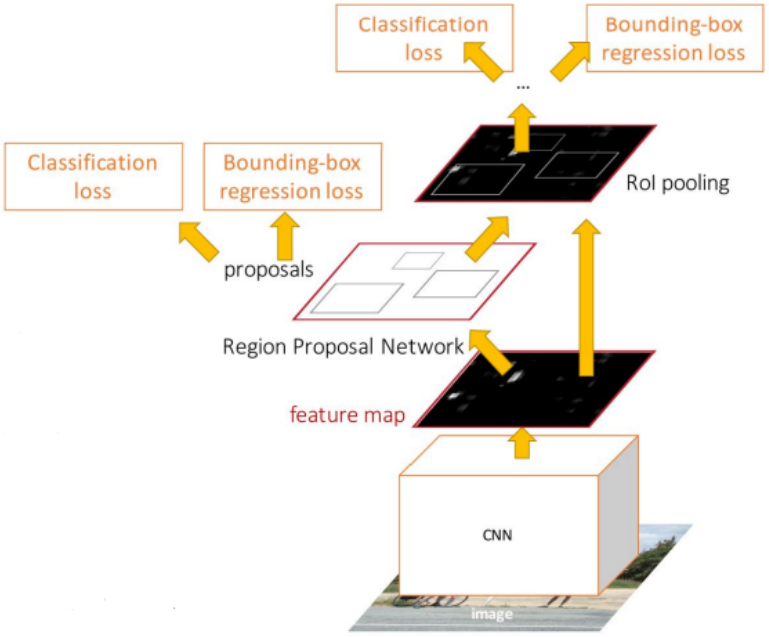


Figure 1: Different schemes for addressing multiple scales and sizes. (a) Pyramids of images and feature maps are built, and the classifier is run at all scales. (b) Pyramids of filters with multiple scales/sizes are run on the feature map. (c) We use pyramids of reference boxes in the regression functions.

The model is jointly trained with 4 losses:

1) RPN classify: "object"/"not object"
2) RPN regress box coordinates
3) Final classification score (object class)
4) Final box coordinates



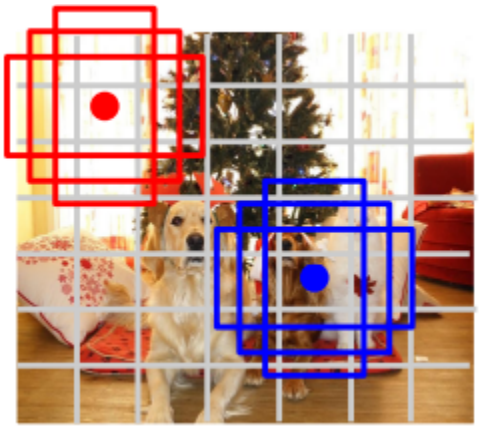
Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

YOLO

Here, A single convolutional network simultaneously predicts multiple bounding boxes and class probabilities for those boxes. YOLO reasons globally about the image when making predictions. Unlike sliding window and region proposal-based techniques, YOLO sees the entire image during training and test time so it implicitly encodes contextual information about classes as well as their appearance.

1) divide the image into a SxS grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. For each grid cell, define B base boxes centered at each grid cell (shifting crops over the grid cell). Each grid cell predicts B bounding boxes and confidence scores for those boxes.

   For example, here B=3:

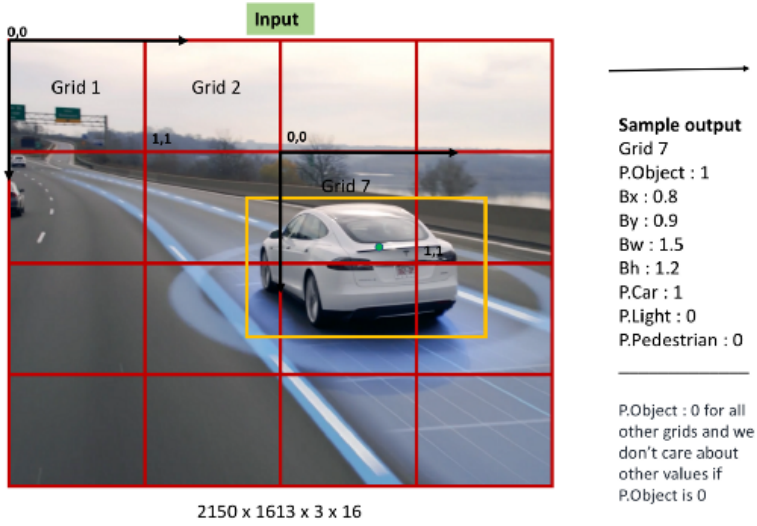


   These confidence scores reflect how confident the model is that the box contains an object and also how accurate it thinks the box is that it predicts.

   Formally we define confidence as Pr(Object) $* IOU^{truth}_{pred}$. If no object exists in that cell, the confidence scores should be zero. Otherwise, we want the confidence score to equal the intersection over union (IOU) between the predicted box and the ground truth.

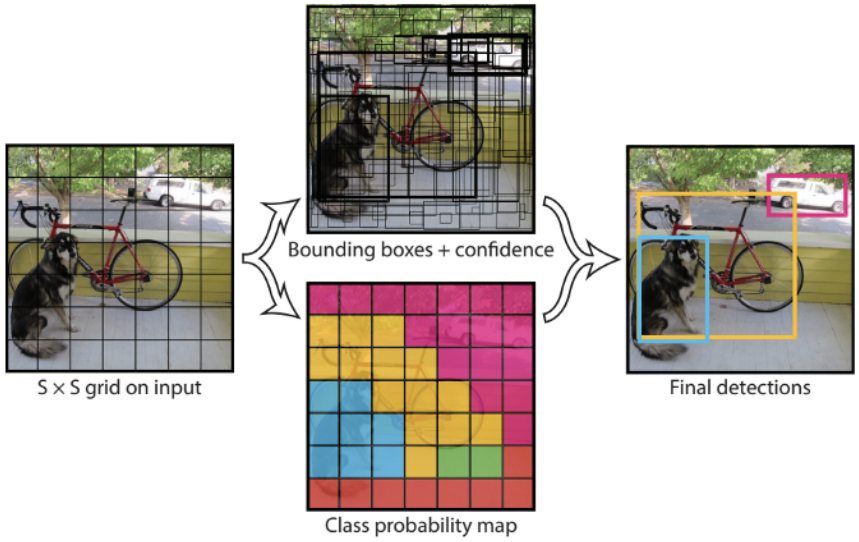
2) Each grid cell predicts B bounding boxes and confidence scores for those boxes: (dx, dy, dh, dw, confidence). The (x, y) coordinates represent the center of the box relative to the bounds of the grid cell. The width and height are predicted relative to the whole image. Finally, the confidence prediction represents the IOU between the predicted box and any ground truth box. (it will help later to calculate the class probability of the grid cell).
3) Each grid cell also predicts C conditional class probabilities. These probabilities are conditioned on the grid cell containing an object. <u>We only predict one set of class probabilities per grid cell, regardless of the number of boxes B.</u>

output example for grid cell number 7:



overall: The output tensor will have the following shape: $7 \times 7 \times (5B + C)$

4) The bounding boxes having the class probability above a threshold value are selected and used to locate the object within the image.



Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

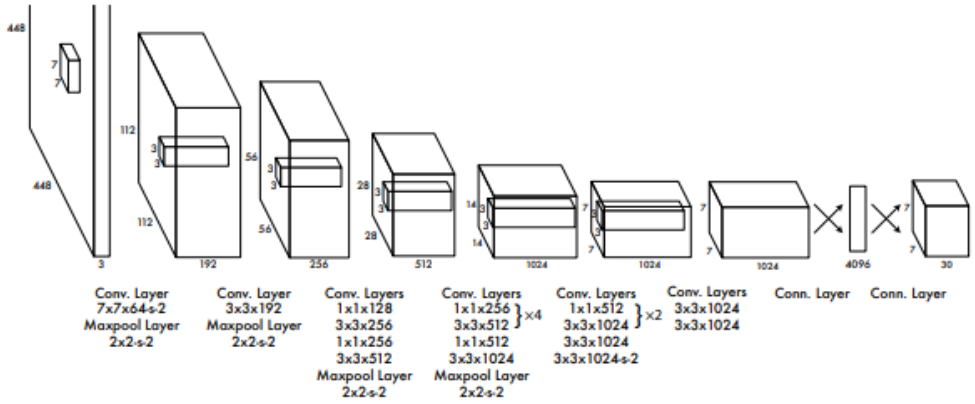


Figure 3: The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

For evaluating YOLO on PASCAL VOC, we use S = 7, B = 2. PASCAL VOC has 20 labeled classes so C = 20. Our final prediction is a 7 × 7 × 30 tensor.

Training:
YOLO predicts multiple bounding boxes per grid cell. At training time we only want one bounding box predictor to be responsible for each object. We assign one predictor to be “responsible” for predicting an object based on which prediction has the highest current IOU with the ground truth. This leads to specialization between the bounding box predictors. Each predictor gets better at predicting certain sizes, aspect ratios, or classes of object, improving overall recall.

loss function:

$$\lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$

$$+ \lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$$

$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$$

$$+ \sum_{i=0}^{S^2} 1_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

where 1_i^{obj} denotes if an object appears in cell i and 1_{ij}^{obj} denote that the jth bounding box predictor in cell i is “responsible” for that prediction.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

FPN

DETR: End-to-End Object Detection with Transformers

Two ingredients are essential for direct set predictions in detection: (1) a set prediction loss that forces unique matching between predicted and ground-truth boxes; (2) an architecture that predicts (in a single pass) a set of objects and models their relation.

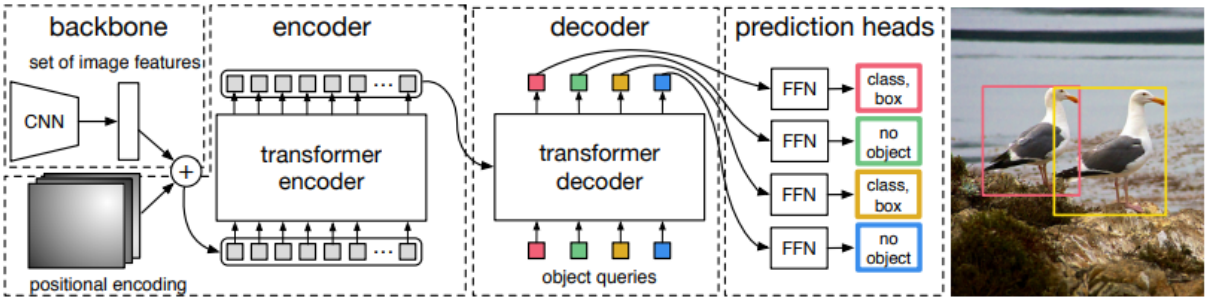


Fig. 2: DETR uses a conventional CNN backbone to learn a 2D representation of an input image. The model flattens it and supplements it with a positional encoding before passing it into a transformer encoder. A transformer decoder then takes as input a small fixed number of learned positional embeddings, which we call *object queries*, and additionally attends to the encoder output. We pass each output embedding of the decoder to a shared feed forward network (FFN) that predicts either a detection (class and bounding box) or a "`no object`" class.

Architecture:

1. A CNN backbone to extract a compact feature representation - starting from the initial image, a conventional CNN backbone generates a l**ower resolution activation map**.
2. Transformer encoder - produce feature maps. First, a 1x1 convolution reduces the channel dimension of the high-level activation map f from C to a smaller dimension d. The encoder expects a sequence as input, hence we collapse the spatial dimensions of the feature map into one dimension, resulting in a d×HW feature map. Each encoder layer has a standard architecture and consists of a multi-head self-attention module and a feed-forward network (FFN). Since the transformer architecture is permutation-invariant, we supplement it with fixed positional encodings that are added to the input of each attention layer. Both query and key elements are of pixels in the feature maps.
3. Transformer decoder - produce object query features. For the Transformer decoder in DETR, the input includes both feature maps from the encoder, and N object queries represented by learnable positional embeddings (e.g., N = 100). There are two types of attention modules in the decoder, namely, cross-attention and self-attention modules. In the cross-attention modules, object queries extract features from the feature maps. The query elements are of the object queries, and key elements are of the output feature maps from the encoder. The difference with the original transformer is that our model decodes the N objects in parallel at each decoder layer, while Vaswani et al. use an autoregressive model that predicts the output sequence one element at a time. Since the decoder is also permutation-invariant,

the N input embeddings must be different to produce different results. These input embeddings are learned positional encodings that we refer to as object queries, and similarly to the encoder, we add them to the input of each attention layer. The N object queries are transformed into an output embedding by the decoder. They are then independently decoded into box coordinates and class labels by a feed-forward network, resulting in N final predictions. Using self- and encoder-decoder attention over these embeddings, the model globally reasons about all objects together using pair-wise relations between them, while being able to use the whole image as context.

4. Feed forward network (FFN) - regression branch that predicts the bounding box coordinates and classification branch to produce the classification results. The final prediction is computed by a 3-layer perceptron with ReLU activation function and hidden dimension d, and a linear projection layer. The FFN predicts the normalized center coordinates, height, and width of the box w.r.t. the input image, and the linear layer predicts the class label using a softmax function. Since we predict a fixed-size set of N bounding boxes, where N is usually much larger than the actual number of objects of interest in an image, an additional special class label ∅ is used to represent that no object is detected within a slot. This class plays a similar role to the "background" class in the standard object detection approaches.

Loss:
DETR infers a fixed-size set of N predictions, in a single pass through the decoder, where N is set to be significantly larger than the typical number of objects in an image. Our loss produces an optimal bipartite matching between predicted and ground-truth objects and then optimize object-specific (bounding box) losses.

y - the ground truth set of objects

$\hat{y}$- the set of N predictions

the loss is a pairwise matching cost between ground truthy_i and a prediction with index σ(i)

$$\hat{\sigma} = \arg\min_{\sigma \in \mathfrak{S}_N} \sum_{i}^{N} \mathcal{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)}),$$

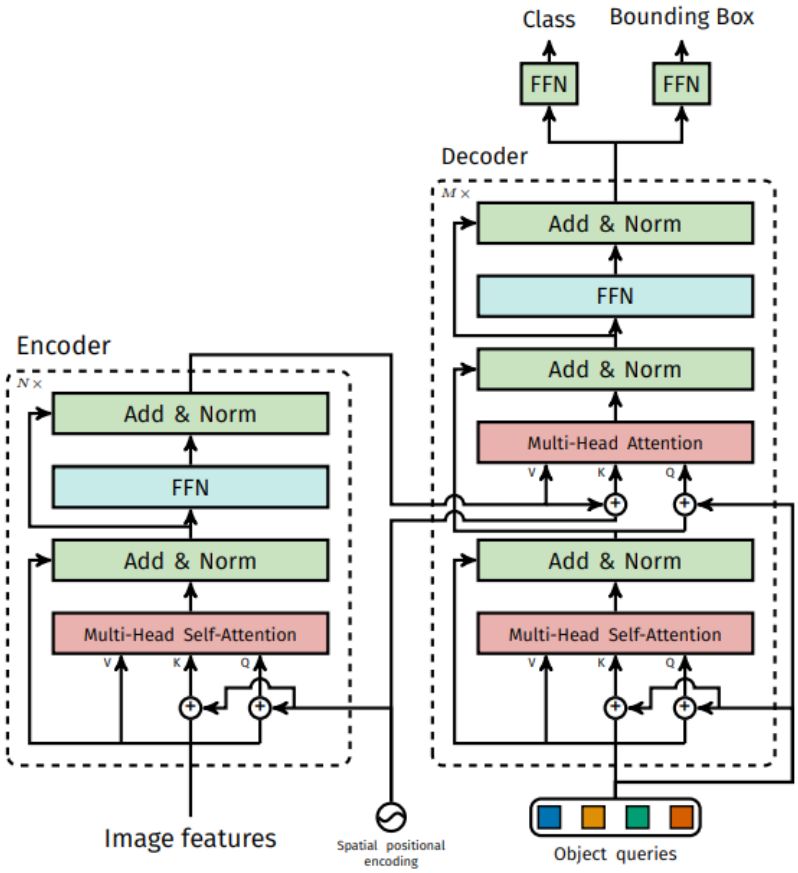
The second step is to compute the loss function, the Hungarian loss for all pairs matched in the previous step. We define the loss similarly to the losses of common object detectors, i.e. a linear combination of a negative log-likelihood for class prediction and a box loss

$$\mathcal{L}_{\text{Hungarian}}(y, \hat{y}) = \sum_{i-1}^{N} \left[-\log \hat{p}_{\hat{\sigma}(i)}(c_i) + \mathbb{1}_{\{c_i \neq \varnothing\}} \mathcal{L}_{\text{box}}(b_i, \hat{b}_{\hat{\sigma}}(i)) \right] ,$$

the bounding box loss:

The most commonly-used `1 loss will have different scales for small and large boxes even if their relative errors are similar. To mitigate this issue we use a linear combination of the `1 loss and the generalized IoU loss.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



Issues with DETR: These issues can be mainly attributed to the deficits of Transformer attention in handling image feature maps as key elements: (1) DETR has relatively low performance in detecting small objects. Modern object detectors use high-resolution feature maps to better detect small objects. However, high-resolution feature maps would lead to an unacceptable complexity for the self-attention module in the Transformer encoder of DETR, which has a quadratic complexity with the spatial size of input feature maps. (2) Compared with modern object detectors, DETR requires many more training epochs to converge.

Instance Segmentation

Driven by the effectiveness of RCNN, many approaches to instance segmentation are based on segment proposals:

1) learn to propose segment candidates, which are then classified by Fast R-CNN. In these methods, segmentation precedes recognition, which is slow and less accurate.
2) combining the segment proposal system and object detection system for “fully convolutional instance segmentation” (FCIS). The common idea is to predict a set of position-sensitive output channels fully convolutionally. These channels simultaneously address object classes, boxes, and masks, making the system fast.

3) Another family of solutions to instance segmentation is driven by the success of semantic segmentation. Starting from per-pixel classification results (e.g., FCN outputs), these methods attempt to cut the pixels of the same category into different instances. In contrast to the segmentation-first strategy of these methods, Mask R-CNN is based on an instance-first strategy.

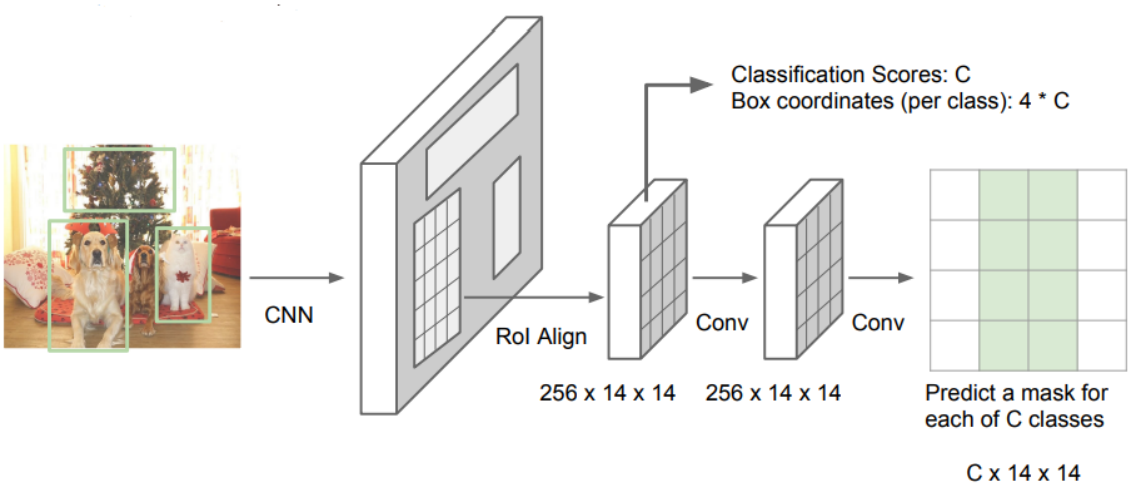
Mask R-CNN

Mask R-CNN extends faster R-VNN by adding a branch for predicting segmentation masks on each region of interest (RoI), in parallel with the existing branch for classification and bounding box regression. The mask branch is a small FCN applied to each RoI, predicting a segmentation mask in a pixel to pixel manner.

The additional mask output is distinct from the class and box outputs, requiring the extraction of a much finer spatial layout of an object.

Mask R-CNN adopts the same two-stage procedure, with an identical first stage (which is RPN). In the second stage, in parallel to predicting the class and box offset, Mask R-CNN also outputs a binary mask for each RoI.

Specifically, for each RoI, we predict C masks, each mask of size m × m using an FCN (where C is the number of classes and mxm is the size of the RoI). This allows each layer in the mask branch to maintain the explicit m × m object spatial layout without collapsing it into a vector representation that lacks spatial dimensions.



- **Application - Homan Pose estimation**

  This framework can easily be extended to human pose estimation. We model a keypoint's location as a one-hot mask and adopt Mask R-CNN to predict K masks, one for each of K keypoint types (e.g., left shoulder, right elbow). We make minor modifications to the segmentation system when adapting it for keypoints:
  For each of the K key points of an instance, the training target is a one-hot m × m binary mask where only a single pixel is labeled as foreground (a.k.a, for each instance the target is K binary masks, as the number of key points). During training, for each visible ground-truth keypoint, we minimize the cross-entropy loss over an m*m -way softmax output (which encourages a single point to be detected).

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Unsupervised learning

unsupervised learning - a type of algorithm that learns patterns/underlying hidden structure from untagged data. Examples: clustering, dimensionality reduction, feature learning, density estimation.
Clustering - find groups in the data that are similar through some type of metric (k means clustering for example)
Dimensionality reduction - we want to find axes along in which our training data has the most variation. These axes are part of the underlying structure of the data and then we can use it to reduce the dimensionality of the data such that the data has significant variation among each of the remaining dimensions.
Learning feature representation for data - we try to reconstruct the data using autoencoder and L2 loss. We use this to learn features.
Density estimation - we want to estimate the underlying distribution of our data. We want to fit a model such that the density is higher where there are more points concentrated. For example generative models.

Generative models

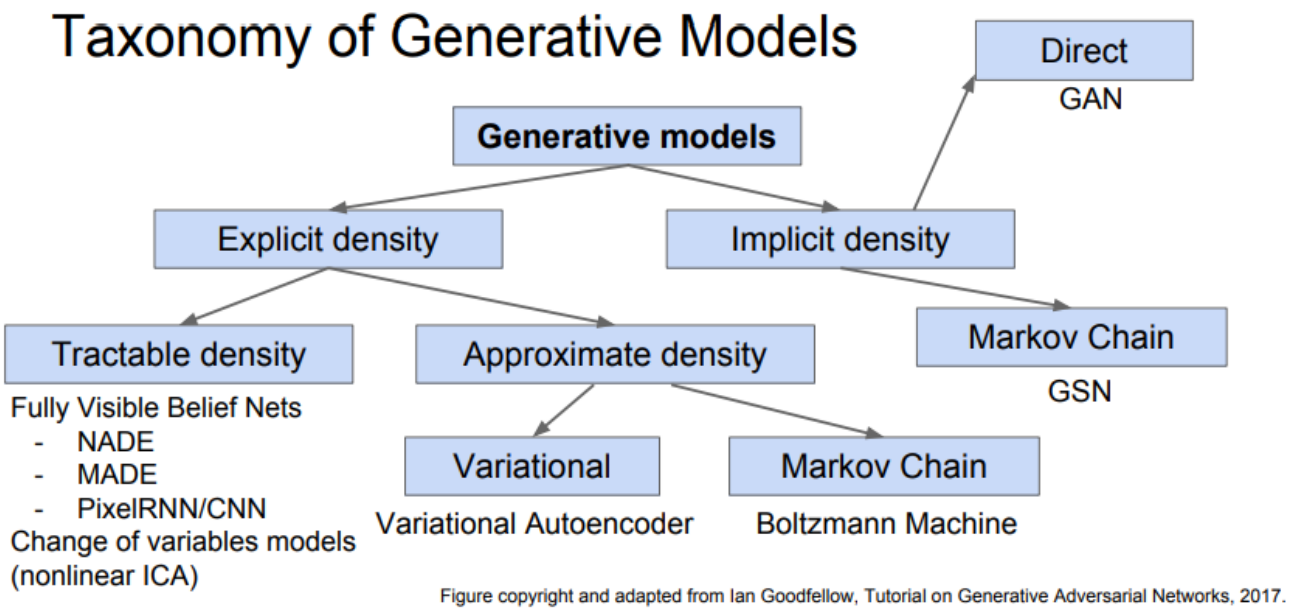


Figure copyright and adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

Definition: class of models for unsupervised learning where given training data, the goal is to generate new samples from the same distribution. Want to learn $p_{model}(x)$ similar to $p_{data}(x)$

Addresses density estimation, a core problem in unsupervised learning.
Several options:

- **Explicit density estimation**: explicitly define and solve for $p_{model}(x)$ (PixelCNN, VAE)
- **Implicit density estimation**: learn a model that can sample from $p_{model}(x)$ w/o explicitly defining it (GAN)

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

PixelCNN/PixelRNN -Explicit density estimation

Key idea: We want to compute the likelihood of an image X. We will use the chain rule to decompose it into a product of 1-d distributions: the probability of each pixel x_i conditioned on all previous pixels x1,...x_(i-1). then the likelihood of all pixels in the image is going to be the product of all these likelihoods together. In order to train this model, we can just maximize the likelihood of our training data.

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

These kinds of models are preferably used in **image completion**.

Figure 1. Image completions sampled from a PixelRNN.

How does pixelRNN work? It generates image pixels starting from the left corner of the image. We will sequentially generate pixels based on the connections of the pixel with its neighbors. The dependency on the previous pixel is modeled using an RNN. It is a sequential generation - thus, very slow!

PixelCNN - still generates image pixels starting from the corner and expanding outwards, but the difference now is that instead of using an RNN to model all these dependencies, we are going to use a CNN over a context region around the particular pixel we are going to generate. For each pixel, we are taking a small area within the region that has already been generated and then passing this through a CNN and using that to generate our next pixel value. For each pixel, the output is a softmax over the pixel values 0-255. we can train this by maximizing the likelihood of the training images - for each training image, we will do the generation process and at each pixel location we have the ground truth value from the training image and this is basically the label

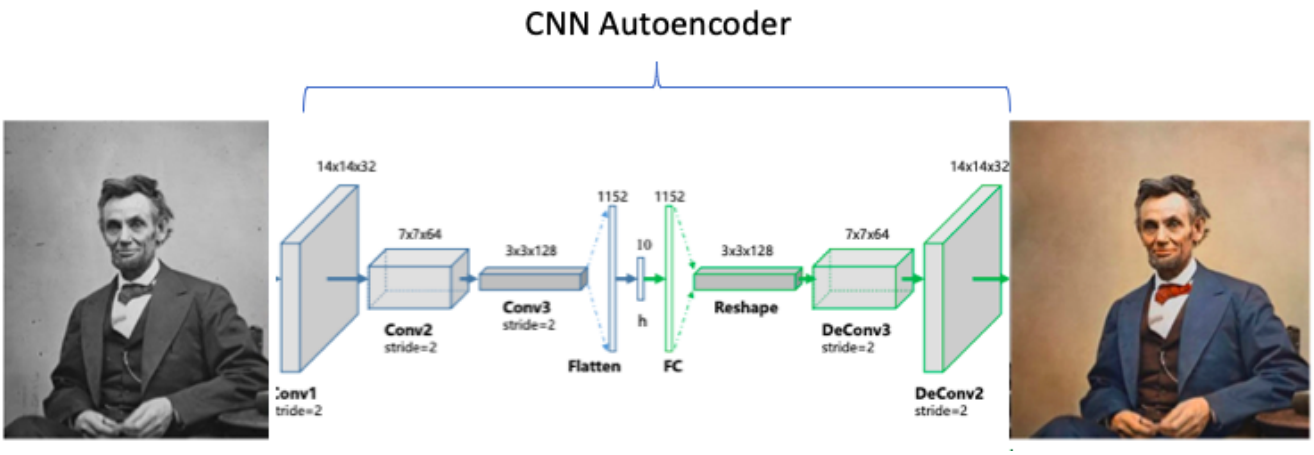
Variational Autoencoders (VAE)

VAEs are based on Autoencoders.

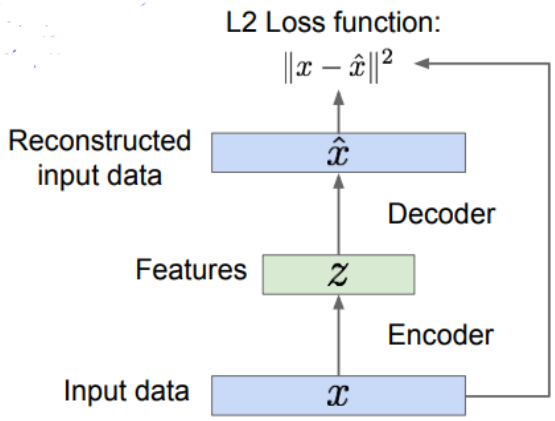
Autoencoder

- Applications: 1) image distortion tasks (as denoising). 2) Learning representation for large unlabeled data and then applying the learned features to a self-supervised task with smaller data.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



Unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data. specifically, we learn feature vector $z$from input data X. For this, we use an encoder that is a function mapping from the input data to the vector z . This encoder is usually a CNN with ReLU non-linearity. Since z is smaller than X, it enables learning features that capture meaningful factors of variation in data. We learn $z$by training an encoder-decoder scheme such that features can be used to reconstruct original data.



Now we can throw away the decoder, and use the encoder to initialize a supervised model. For example, we can take smaller input data (a subset of labels), extract the features, and then have an additional classifier network on top of this to output class labels - we will fine-tune the encoder jointly with the classifier. These features can capture factors of variation in training data.

VAE is a probabilistic spin on autoencoders that will let us sample from the model in order to generate new data.

We assume that our training data$(x_1,..\, x_N)$is generated from underlying unobserved (latent) representation z. z is some vector that each coordinate of z captures how much factor of variation we have in the training data - the elements of z can be different kinds of attributes. For example, if we want to generate faces, it could be how much of a smile is on the face, the position of the hair, the orientation of the face. All these are examples of latent factors that can be learned. $z^1 - \ color\ hair,\ z^2 - \ smile,\ z^3 - \ orientation\ of\ face$ ...

The generation process:

- we are going to sample from a prior over Z (for example, for each of these attributes, we can have a prior over what sort of distribution we think that there should be for this (a gaussian is natural prior we can use to each of the factors of Z))
- We are going to generate our data X by sampling from a conditional distribution. we sample Z first, we sample a value for each of the latent factors, and then we will use that and sample our image X from there.

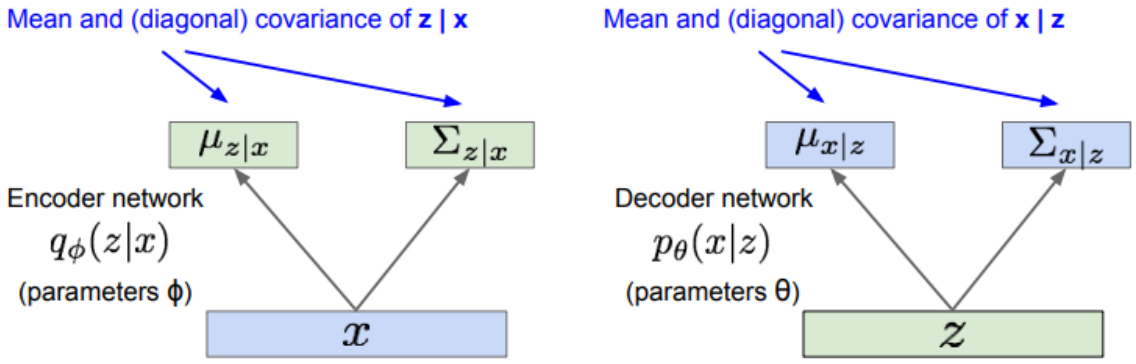
In order to generate new images, we want to estimate the true parameters θ^* of our prior and conditional distributions.

Just as a standard autoencoder, a variational autoencoder is an architecture composed of both an encoder and a decoder and that is trained to minimize the reconstruction error between the encoded-decoded data and the initial data. However, in order to introduce some regularization of the latent space, we proceed to a slight modification of the encoding-decoding process: instead of encoding an input as a single point, we encode it as a distribution over the latent space.

How should we represent the model?
choose prior p(z) to be simple - e.g. gaussian.

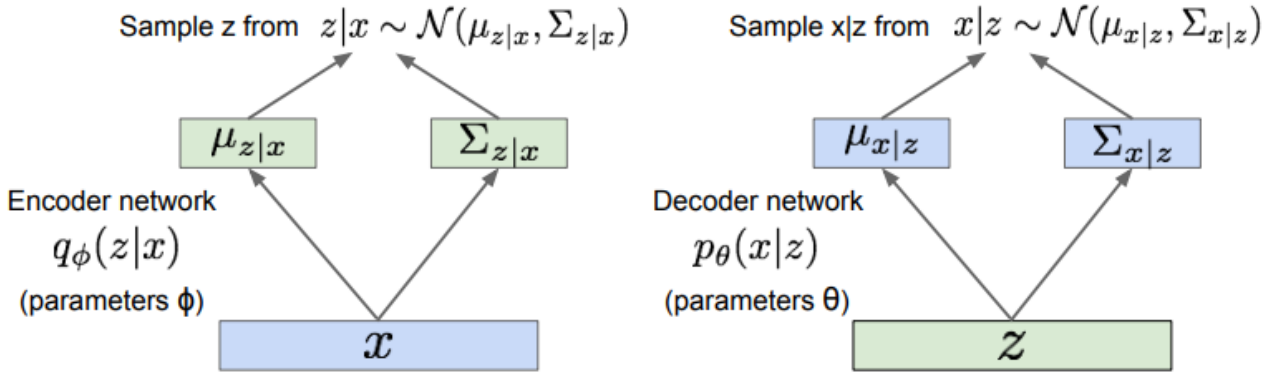
- Generating an image from the latent vector ($p_\theta(x|z)$) - Now we want to generate an image using the conditional distribution p(x/z). We will do this by using a decoder network that takes some latent representation and decodes it to the training image.
- Producing latent vector from training image ($q_\phi(z|x)$) - define additional encoder network that approximates $p_\theta(z|x)$. This allows us to derive a lower bound on the data likelihood that is traceable, which we can optimize.

The output of the two networks are mean and covariance (these two are probabilistic networks!)



Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

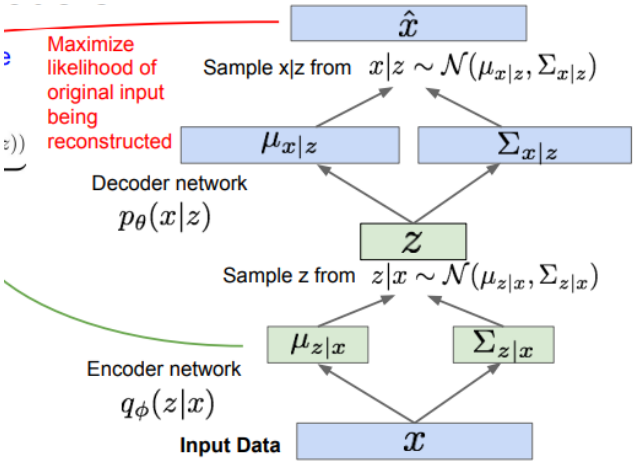
In order to get z given x and x given zWe sample from the distribution the networks produce. The encoder and decoder networks are producing distributions over$z$ and x respectively and we sample from these distributions.



Training:

Input: training data X.

1) Pass it through the encoder network and get $q_\phi(z|x)$. Here, the input is encoded as a distribution over the latent space.
2) From here, sample z from this distribution (intuitively - $z$represents the factors that we can infer from$x$). Here, a point from the latent space is sampled from that distribution.
3) Then, pass z through the second decoder network, and from the decoder network get the mean and variance of the distribution x given$z$.
4) Using the mean and the variance from 3), sample now x given z from this distribution (this is $\hat{x}$). The sampled point is decoded and the reconstruction error can be computed

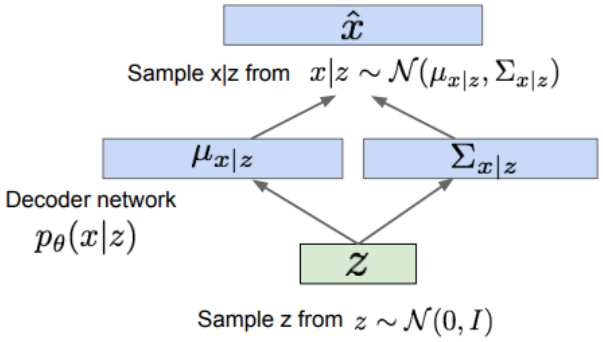


Training loss

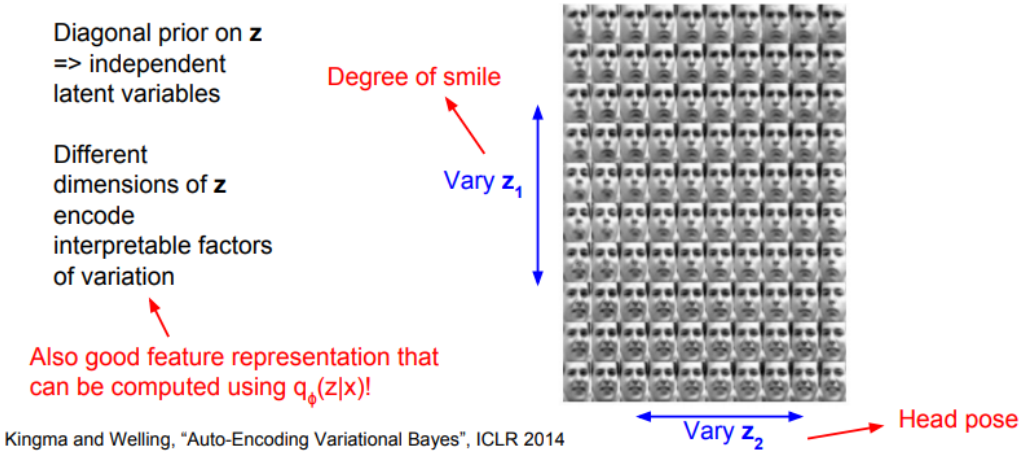
The encoded distributions are chosen to be normal so that the encoder can be trained to return the mean and the covariance matrix that describes these Gaussians. The reason why an input is encoded as a distribution with some variance instead of a single point is that it makes it possible to express very naturally the latent space regularisation: the distributions returned by the encoder are enforced to be close to a standard normal distribution.
Thus, the loss function that is minimized when training a VAE is composed of a "reconstruction loss" (on the final layer), that tends to make the encoding-decoding scheme as performant as possible, and a "regularisation term" (on the latent layer), that tends to regularise the organization of the latent space by making the distributions returned by the encoder close to a standard normal distribution.

Generating new images - once we have trained our VAE, will use just the decoder to generate new samples. we sample z from the prior (regular gaussian - we chose z to be sampled from diagonal distribution), and then we can sample data x using the output distribution of the decoder (with the learned distribution)
if we vary the specific element of z (fixing all elements and changing one element) we see transition smoothly.

Use decoder network. Now sample z from prior!



Why diagonal prior? In order to encourage the decoder to output independent latent variables that can encode interpretable factors of variations, as we vary one element we see the attribute changing (for example the smile is changing) → **good for editing** !!!!



How can we use the encoder? z variables are good feature representations because they encode interpretable factors of variation. So, we can take this pretrained encoder network, and give it an input images X, we can map this to z and use the z as features

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

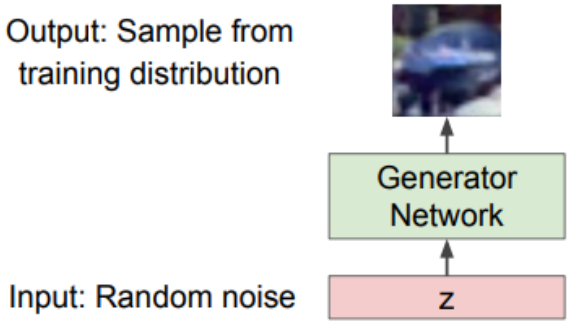
Summary

VAEs are known to give representations with disentangled factors. This happens due to isotropic Gaussian priors on the latent variables. Modeling them as Gaussians allows each dimension in the representation to push themselves as farther as possible from the other factors. Also, added a regularization coefficient that controls the influence of the prior. Prior gives significant control over how we want to model our latent distribution. This kind of control does not exist in the usual AE framework. This is actually the power of Bayesian models themselves, VAEs are simply making it more practical and feasible for large-scale datasets. So, to conclude, if you want precise control over your latent representations and what you would like them to represent, then choose VAE.

GAN - generative adversarial networks

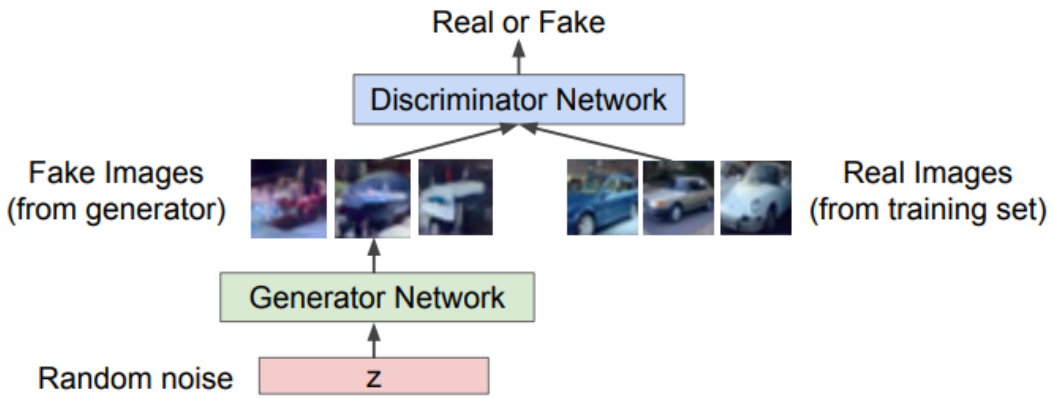
GANs are generative models that learn a mapping from random noise vector z to output image y, $G\colon\ z \rightarrow y$. Here, the goal is to sample from complex, high-dimensional training distribution. Rather than sampling from this complex distribution, we will sample from a simple distribution, e.g. random noise, and learn transformation from the simple distribution to the training distribution.

Intuition: Assume that for each image in the distribution of images there is a unique random noise. Now, consider the noise vector of size 128. In addition, we assume (for this example) that the first entry of the vector corresponds to the attribute “length of hair on the head”. After training, the model has learned that for “bald” the value in the first entry is 0, for long hair the value is 1, and by selecting a random number from 0 to 1 the amount of hair is determined. So, the model can generate persons with different hair lengths.
In this way, all 128 entries in random noise will decide on a specific human face. That's why every time choosing random noise will generate a new person image. In addition, if you use the same random noise every time, then the model will generate the same image.



Two-player game:
Generator network: try to fool the discriminator by generating real-looking images
Discriminator network: try to distinguish between real and fake images

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



Train jointly in **minimax game**

θ_g – *parameters of network G*

$log\ D_{\theta_d}(x)$ – *likelihood of real data being real from the data distribution* P_{data}

Train jointly in **minimax game**

Discriminator outputs likelihood in (0,1) of real image

Minimax objective function:

$$\min_{\theta_g}\max_{\theta_d}\left[\mathbb{E}_{x\sim p_{data}}\log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data x}} + \mathbb{E}_{z\sim p(z)}\log(1-\underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data G(z)}})\right]$$

- Discriminator (θ_d) wants to **maximize objective** such that D(x) is close to 1 (real) and D(G(z)) is close to 0 (fake)
- Generator (θ_g) wants to **minimize objective** such that D(G(z)) is close to 1 (discriminator is fooled into thinking generated G(z) is real)

Examples of GANs:

DCGAN

Generator: ConvTranspose -> BN -> ReLU -> .. -> ConvTranspose -> BN -> ReLU -> ConvTranspose -> Tanh

Tanh since the data is normalized to be in (-1,1) at the preprocessing step

Discriminator: Conv -> LeakyReLU -> Conv -> BN -> LeakyReLU -> .. Conv -> BN -> LeakyReLU -> ConV -> Sigmoid

Sigmoid since we use BCELoss (want the values of the discriminator to be in range (0,1))

Key ideas:

1) They replace all the spatial pooling functions (as max-pooling) with strided convolutions, allowing the network to learn its own spatial downsampling. In the same way, all the “unpooling” layers with transpose convolution.

2) Eliminate fully connected layers on top of convolutional features, and replace it with global Average Pooling.

3) Use batchNorm in both the generator and the discriminator.

4) Use ReLU activation in the generator for all layers except for the output, which uses Tanh (since they preprocess the values of the images to be [-1,1].

5) Use LeakyReLU activation in the discriminator for all layers. The last convolution layer in the discriminator is flattened and then fed into a single sigmoid output.

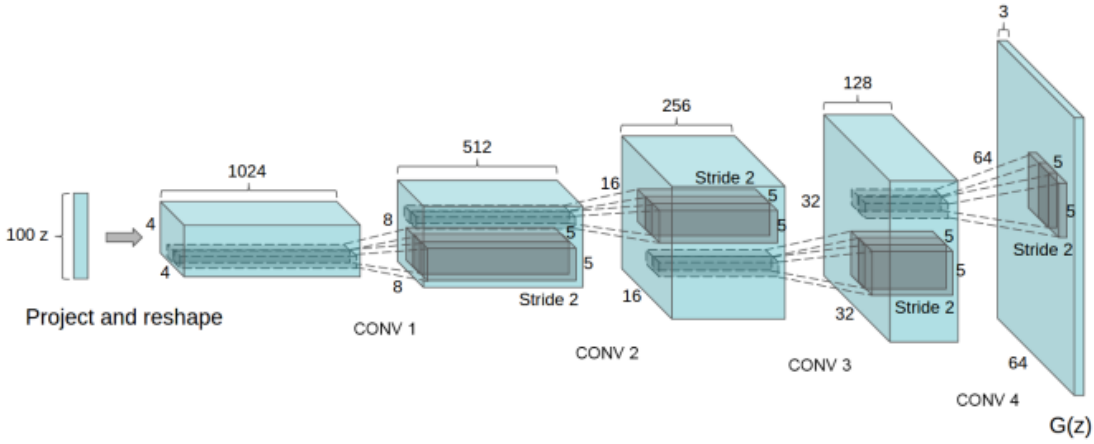


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions) then convert this high level representation into a 64×64 pixel image. Notably, no fully connected or pooling layers are used.

Architecture:

Input to the generator: noise of shape: (batc_size, 100, 1, 1)

```
Generator(
 (main): Sequential(
  (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU(inplace=True)
  (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (5): ReLU(inplace=True)
  (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (8): ReLU(inplace=True)
  (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (11): ReLU(inplace=True)
  (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (13): Tanh()
 )
```

The discriminator, D, is a binary classification network that takes an image as input and outputs a scalar probability that the input image is real (as opposed to fake). Here, D takes a 3x64x64 input image, processes

it through a series of Conv2d, BatchNorm2d, and LeakyReLU layers, and outputs the final probability through a Sigmoid activation function.

```
Discriminator(
 (main): Sequential(
  (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (1): LeakyReLU(negative_slope=0.2, inplace=True)
  (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (4): LeakyReLU(negative_slope=0.2, inplace=True)
  (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (7): LeakyReLU(negative_slope=0.2, inplace=True)
  (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (10): LeakyReLU(negative_slope=0.2, inplace=True)
  (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (12): Sigmoid()
 )
```

Loss:

In DCGAN they define the real label as 1 and fake label as 0. Thus, we want to optimize the discriminator to output "1" for the real images, and 0 for the fake images. In addition, we want to optimize the generator to output "1" for the fake images.

Since we formalize this as a classification problem with 2 classes, we can use the BCE loss.

```
criterion = nn.BCELoss()
errD_real = criterion(output_netD_real , label_real)
errD_real.backward()
errD_fake = criterion(output_netD_fake, label_fake)
errD_fake = criterion(output, label)
```

Conditional GAN (image to image mapping)

GANs are generative models that learn a mapping from random noise vector z to output image y, $G: z \rightarrow y$. In contrast, conditional GANs learn a mapping from observed image $x$and random noise vector z to y, G : {x, z} → y. In a simple conditional adversarial net, the input noise is concatenated to the condition (can be image or class) before feeding to the generator, and similarly in the discriminator - the output image of the generator/the real image is concatenated to the condition before passed to the discriminator.

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x}|\boldsymbol{y})] + \mathbb{E}_{\boldsymbol{z}\sim p_z(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}|\boldsymbol{y})))].$$

Pix2Pix: Image-to-Image Translation with Conditional Adversarial Networks

In this paper, they explore GANs in the conditional setting. Just as GANs learn a generative model of data, conditional GANs (cGANs) learn a conditional generative model. This makes cGANs suitable for image-to-image translation tasks, where we condition on an input image and generate a corresponding

output image

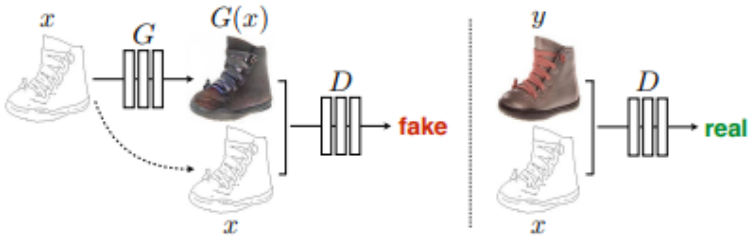


Figure 2: Training a conditional GAN to map edges→photo. The discriminator, D, learns to classify between fake (synthesized by the generator) and real {edge, photo} tuples. The generator, G, learns to fool the discriminator. Unlike an unconditional GAN, both the generator and discriminator observe the input edge map.

Simply, the condition is an image and the output is another image. **Both generator and discriminator use modules of the form convolution-BatchNorm-ReLu.**

For the generator, they use the **U-net generator** - the design of the generator is based on the assumption that both input-output images are renderings of the same underlying structure (e.g edges). This means that there is low-level information that is shared, hence symmetrical skip connections are implemented between the encoder and the decoder of the generator, in an unet style architecture.

For the discriminator, they use a convolutional "**PatchGAN**" classifier, which only penalizes structure at the scale of image patches.

Furthermore, they provided an extra L1 loss between the generated image and the ground truth. In this way, they argue that the image quality will be higher since L1 loss leads to less blurring than L2 loss.

The final objective is:

$$G^* = \arg\min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G).$$

where:

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x, z)\|_1].$$

$$\mathcal{L}_{cGAN}(G, D) = \mathbb{E}_{x,y}[\log D(x, y)] + \mathbb{E}_{x,z}[\log(1 - D(x, G(x, z))],$$

- Note: Without z, the net could still learn a mapping from x to y, but would produce deterministic outputs, and therefore fail to match any distribution other than a delta function.
- They provide noise to the generator only in the form of dropout, applied on several layers of the generator at both training and test time. **However, this method does not provide great variability.**

Unet

Motivation: A defining feature of image-to-image translation problems is that they map a high-resolution input grid to a high-resolution output grid. In addition, for the problems we consider, the input and output differ in surface appearance, but both are renderings of the same underlying structure. Therefore, the

structure in the input is roughly aligned with the structure in the output. We design the generator architecture around these considerations. For this, they add skip connections between each layer i and layer n − i, where n is the total number of layers. Each skip connection simply concatenates all channels at layer i with those at layer n − i.

Markovian discriminator PatchGAN

L2 loss – and L1, produce blurry results on image generation problems. Although these losses fail to encourage high-frequency crispness, in many cases, they accurately capture the low frequencies.

This motivates restricting the GAN discriminator to only model a high-frequency structure, relying on an L1 term to force low-frequency correctness. As stated, modeling high-frequencies is related to local information that can be found in small image sub-regions that we call patches.

The discriminator only penalizes structure at the scale of patches. This discriminator tries to classify if each 70 ×70 patch in an image is real or fake. The output of the discriminator is a feature map of size 30x30. Each value in the map corresponds to a different patch in the image with size 70x70. Thus, each value classifies the corresponding patch as “real” or “fake”.

In case of BCE loss: the shape of the discriminator output is (30x30) and the shape of the target is also 30x30. If the discriminator gets real images, all the values in the target tensor are “1”, otherwise “0”.

In case of Wasserstein Gan Loss: Similarly, the shape of the discriminator output is (30x30). if the target is “real” then we want to maximize the mean of this tensor (-prediction.mean()), otherwise we want to minimize the mean of this tensor (prediction.mean()).

CycleGAN - image to image translation

Note that Pix2Pix requires training pairs - the input to the discriminator for example is a pair of source image and target image. As opposed to Pix2Pix, CycleGAN learns a mapping from a set (collection) of images to another set of images without paired examples. The models are trained in an unsupervised manner using a collection of images from the source and target domain that do not need to be related in any way.

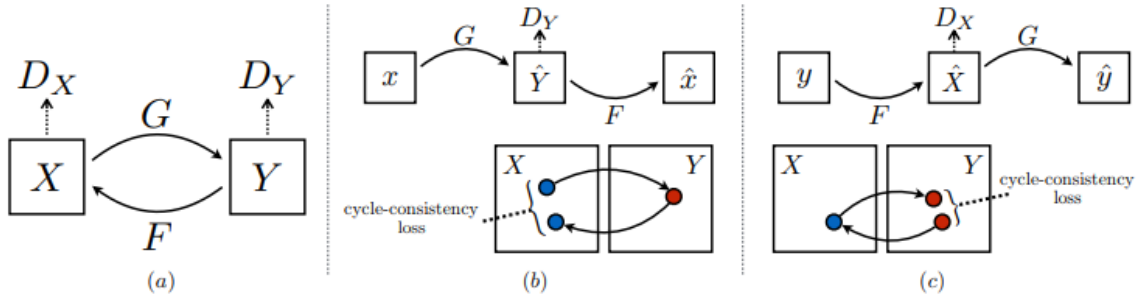


Figure 3: (a) Our model contains two mapping functions $G : X \rightarrow Y$ and $F : Y \rightarrow X$, and associated adversarial discriminators D_Y and D_X. D_Y encourages G to translate X into outputs indistinguishable from domain Y, and vice versa for D_X and F. To further regularize the mappings, we introduce two *cycle consistency losses* that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$

CycleGAN is a Generative Adversarial Network (GAN) that uses two generators and two discriminators. one generator G converts images from the X domain to the Y domain. The other generator is called F, and converts images from Y to X.

Discriminator models are then used to determine how plausible the generated images are and update the generator models accordingly:

$$G : X \rightarrow Y \quad D_y : \text{Distinguishes y from G(x)}$$
$$F : Y \rightarrow X \quad D_x : \text{Distinguishes x from F(y)}$$

There are two adversarial losses: Both generators are attempting to "fool" their corresponding discriminator into being less able to distinguish their generated images from the real versions.

$$Loss_{adv}(G, D_y, X) = \frac{1}{m}\sum_{i=1}^{m}(1 - D_y(G(x_i)))^2 \qquad Loss_{adv}(F, D_x, Y) = \frac{1}{m}\sum_{i=1}^{m}(1 - D_x(F(y_i)))^2$$

However, the adversarial loss alone might be enough to generate plausible images in each domain, but not sufficient to generate translations of the input images. It enforces that the generated output to be of the appropriate domain, but does not enforce that the input and output are recognizably the same. For example, a generator that outputs an image y that was an excellent example of that domain, but looked nothing like x, would do well by the standard of the adversarial loss, despite not giving us what we really want.

The cycle consistency loss addresses this issue. It relies on the expectation that if you convert an image to the other domain and back again, by successively feeding it through both generators, you should get back something similar to what you put in. It enforces that F(G(x)) ≈ x and G(F(y)) ≈ y.

$$Loss_{cyc}(G, F, X, Y) = \frac{1}{m}\sum_{i=1}^{m}[F(G(x_i)) - x_i] + [G(F(y_i)) - y_i]$$

- cycle loss - L1 Loss.

Model Architecture

Each GAN has a conditional generator model that will synthesize an image given an input image. And each GAN has a discriminator model to predict how likely the generated image is to have come from the target image collection.

We can summarize the generator and discriminator models from GAN 1 as follows (and similarly for GAN2):

Generator Model 1:

- Input: Takes photos of summer (collection 1) -> condition!
- Output: Generates photos of winter (collection 2).

Discriminator Model 1:

- Input: Takes photos of winter from collection 2 and output from Generator Model 1.
- Output: Likelihood of image is from collection 2.

Wasserstein GAN

The core idea is to effectively measure how close the model distribution is to the real distribution.

Wasserstein distance:

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma}\left[\, \|x - y\| \,\right],$$

The Wasserstein distance is the minimum cost of transporting mass in converting the real data distribution (Pr) to the generated data distribution (Pg).

This is the same as (from some mathematical duality):

$$W(\mathbb{P}_r, \mathbb{P}_\theta) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim \mathbb{P}_r}[f(x)] - \mathbb{E}_{x \sim \mathbb{P}_\theta}[f(x)]$$

where sup is the least upper bound and f is a 1-Lipschitz function. So to calculate the Wasserstein distance, we just need to find a 1-Lipschitz function. Like other deep learning problems, we can build a deep network to learn it.

In practice, it is very similar to the regular GAN network - the generator is the same and the discriminator is the same except that there is no sigmoid in the discriminator network - the discriminator outputs a scalar score rather than a probability. This score can be interpreted as how real the input images are.

$$L = \underbrace{\mathop{\mathbb{E}}_{\tilde{\boldsymbol{x}} \sim \mathbb{P}_g}[D(\tilde{\boldsymbol{x}})] - \mathop{\mathbb{E}}_{\boldsymbol{x} \sim \mathbb{P}_r}[D(\boldsymbol{x})]}_{\text{Original critic loss}} + \underbrace{\lambda \mathop{\mathbb{E}}_{\hat{\boldsymbol{x}} \sim \mathbb{P}_{\hat{\boldsymbol{x}}}}\left[(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1)^2\right]}_{\text{Our gradient penalty}}.$$

The first term in the loss corresponds to the Wasserstein distance above - we want to minimize the distance between the two distributions. The second term enforces the Lipschitz constraint (A differentiable function f is 1-Lipschitz if and only if it has gradients with norm at most 1 everywhere.)

Two significant contributions for WGAN are: it has no sign of mode collapse in experiments, and
The generator can still learn when the critic performs well.

In PyTorch:

- optimizing the discriminator:

*loss_D = -torch.mean(discriminator(real_imgs)) + torch.mean(discriminator(fake_imgs)) + lambda*gradienet_penalty*
loss_D.backward()

- optimizing the generator:

loss_G = -torch.mean(discriminator(gen_imgs))
loss_G.backward()

ProGAN - high-quality large images

The key innovation of ProGAN is the progressive training — it starts by training the generator and the discriminator with a very low-resolution image (e.g. 4×4) and adds a higher resolution every time by adding layers to the generator and the discriminator.
This technique first creates the foundation of the image by learning the base features which appear even in a low-resolution image and learns more and more details over time as the resolution increases.
To do this, they first artificially shrunk their training images to a very small starting resolution (only 4x4 pixels). They created a generator with just a few layers to synthesize images at this low resolution, and a corresponding discriminator of mirrored architecture. Because these networks were so small, they trained relatively quickly and learned only the large-scale structures visible in the heavily blurred images.

When the first layers finish training, they then add another layer to G and D, doubling the output resolution to 8x8. The trained weights in the earlier layers were kept, but not locked, and the new layer was faded in gradually to help stabilize the transition. Training resumed until the GAN was once again synthesizing convincing images, this time at the new 8x8 resolution.
In this way, they continued to add layers, double the resolution, and train until the desired output size was reached.
The generator architecture for a given resolution k follows a familiar high-level pattern (as In DCGAN): each set of layers doubles the spatial size, and halves the number of channels until the output layer creates an image with just three channels corresponding to RGB. The discriminator does almost exactly the opposite, halving the spatial size and doubling the number of channels with each set of layers.
DCGAN uses transpose convolutions to change the representation size. In contrast, ProGAN uses nearest neighbors for upscaling and average pooling for downscaling. These are simple operations with no learned parameters. They are then followed by two convolutional layers.

- They initialize a small generator and discriminator and pass them a small image. After the training is finished, they add the same generator and discriminator several layers and continue training with the learned weights.

BigGAN

Diffusion Models

Given a sample from the data distribution $x_0 \sim q(x_0)$, produce a markov chain of latent variables x_1, …, x_T by progressively adding gaussian noise to the sample:

$$q(x_t|x_{t-1}) \coloneqq \mathcal{N}(x_t; \sqrt{\alpha_t}x_{t-1}, (1-\alpha_t)\mathcal{I})$$

If the magnitude $1 - \alpha_t$ of the noise added at each step is small enough, the posterior $q(x_{t-1}|x_t)$ is well approximated by a diagonal gaussian. Furthermore, if the magnitude 1 − α1...αT of the total noise added throughout the chain is large enough, x_T is well approximated by N (0, I). These properties suggest learning a model $p_\theta(x_{t-1}|x_t)$ to approximate the true posterior:

$$p_\theta(x_{t-1}|x_t) \coloneqq \mathcal{N}(\mu_\theta(x_t), \Sigma_\theta(x_t))$$

which can be used to produce samples $x_0 \sim p_\theta(x_0)$ by starting with Gaussian noise $x_T \sim N(0, I)$ and gradually reducing the noise in a sequence of steps x_{T-1}, x_{T-2},…, x_0.
To do this, we generate samples $x_t \sim q(x_t|x_0)$ by applying Gaussian noise to x_0, then train a model ε_θ to predict the added noise using a standard mean-squared error loss:

$$L_{\text{simple}} \coloneqq E_{t\sim[1,T],x_0\sim q(x_0),\epsilon\sim\mathcal{N}(0,\mathbf{I})}[||\epsilon - \epsilon_\theta(x_t, t)||^2]$$

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Image restoration

When we are referring to image restoration problems we basically mean that we have a degraded image and we want to recover the clean non-degraded image

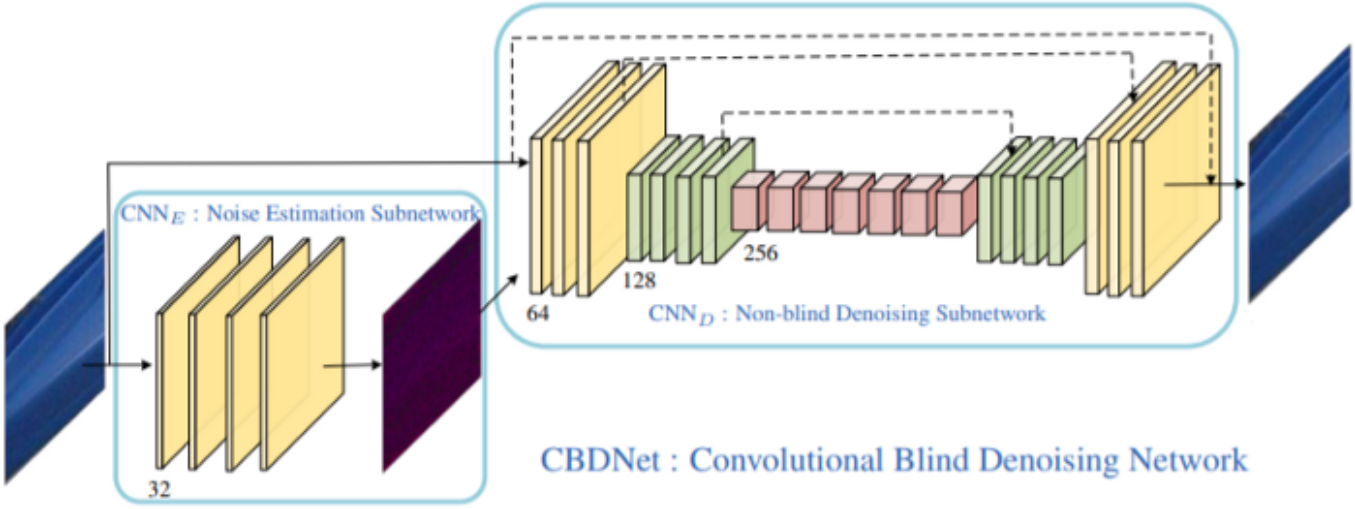
- deep image prior paper (TODO)

Denoising and general reconstruction

Performance metric: PSNR

Baseline - Auto Encoder. Denoising autoencoders corrupt the input image and require the network to undo the damage.

CBDNet - convolutional Blind Denoising Network (2019)



Motivation: existing CNN denoisers tend to be over-fitted to Gaussian noise and generalize poorly to real-world noisy images with more sophisticated noise.

- When trained with a realistic noise model, the memorization ability of CNN will be helpful to make the learned model generalize well to real photographs. Thus, they first generate synthetic noisy images
- noise estimation network - takes a noisy observation y to produce the estimated noise level map A. For this, the network is fully convolutional without pooling and batch normalization. In each convolution (Conv) layer, the number of feature channels is set as 32, and the filter size is 3 × 3. The ReLU nonlinearity is deployed after each Conv layer. We let the output of CNNE be the noise level map due to that it is of the same size as the input y.
- non-blind denoising network - takes as input both y (the noisy image) and the noise level map A, and outputs the final denoising result. The network has the U-net architecture which takes both inputs to produce a clean image. The goal of this network is first to learn the residual mapping R between y and A, and then predict the clean image x = y + R.

symmetric skip connections, strided convolutions, and transpose convolutions are introduced for exploiting multi-scale information as well as enlarging the receptive field. All the filter size is 3×3, and the ReLU nonlinearity is applied after every Conv layer except the last one.

Taking both noisy image and noise level map as input is helpful in generalizing the learned model to images beyond the noise model and thus benefits blind denoising.

Loss for synthetic noise image: noise_estimation_loss + reconstruction_loss + lambda*TV.
noise_estimation_loss = the MSE between the noise level map and the noise variance of the ground truth image (since the image is synthetic, they denoised it with a known variance). The reconstruction loss is MSE between the denoised image and the GT clean image and TX is a regularizer to constrain the smoothness of the noise level map (the L2 regularization).
For the real noisy images, the loss is the reconstruction_loss + lambda*TV due to the unavailability of ground truth noise level map.

Super-resolution

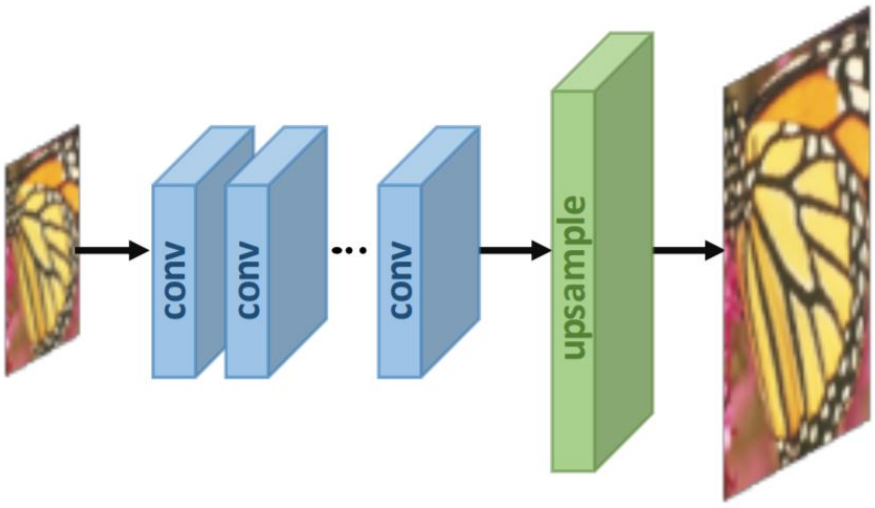
Image super-resolution (SR) is the process of recovering high-resolution (HR) images from low-resolution (LR) images.

Upsampling methods

1) Interpolation based methods
    - Nearest-neighbor Interpolation – The nearest-neighbor interpolation is a simple and intuitive algorithm. It selects the value of the nearest pixel for each position to be interpolated regardless of any other pixels.
    - Bilinear Interpolation – The bilinear interpolation (BLI) first performs linear interpolation on one axis of the image and then performs on the other axis. Since it results in a quadratic interpolation with a receptive field-sized 2 × 2, it shows much better performance than nearest-neighbor interpolation while keeping a relatively fast speed.
    - Bicubic Interpolation – Similarly, the bicubic interpolation (BCI) performs cubic interpolation on each of the two axes Compared to BLI, the BCI takes 4 × 4 pixels into account, and results in smoother results with fewer artifacts but much lower speed.
2) Learning-based upsampling
    - Transposed convolution

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Super-resolution frameworks



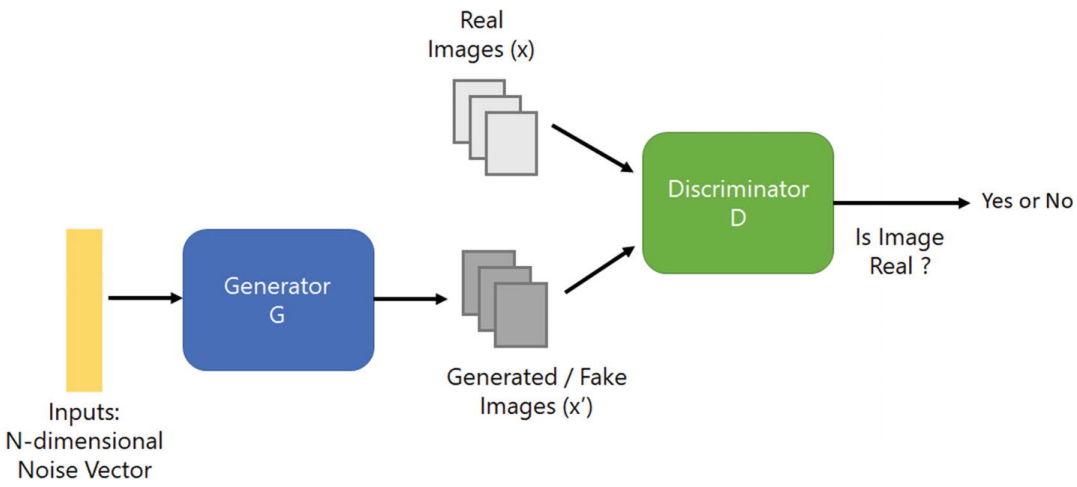
Learning strategies

- Pixelwise L1 loss – Absolute difference between pixels of ground truth HR image and the generated one.
- Pixelwise L2 loss – Mean squared difference between pixels of ground truth HR image and the generated one.
- Content loss – the content loss is indicated as the Euclidean distance between high-level representations of the output image and the target image. High-level features are obtained by passing through pre-trained CNNs like VGG and ResNet.
- Adversarial loss – Based on GAN where we treat the SR model as a generator, and define an extra discriminator to judge whether the input image is generated or not.
- PSNR – Peak Signal-to-Noise Ratio (PSNR) is a commonly used objective metric to measure the reconstruction quality of a lossy transformation. PSNR is inversely proportional to the logarithm of the Mean Squared Error (MSE) between the ground truth image and the generated image.

Super-Resolution Generative Adversarial Network (SRGAN)

Uses the idea of GAN for super-resolution task i.e. generator will try to produce an image from noise which will be judged by the discriminator. Both will keep training so that generator can generate images that can match the true training data.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



There are various ways for super-resolution but there is a problem – how can we recover finer texture details from a low-resolution image so that the image is not distorted?
The results have high PSNR means have high-quality results but they are often lacking high-frequency details.

To achieve this in SRGAN, we use the perceptual loss function which comprises content and adversarial loss.

Steps –

1. We process the HR (high-resolution images) to get downsampled LR images. Now we have HR and LR images for the training dataset.

2. We pass LR images through a generator that upsamples and gives SR (super-resolution) images.

3. We use the discriminator to distinguish HR image and backpropagate GAN loss to train discriminator and generator.

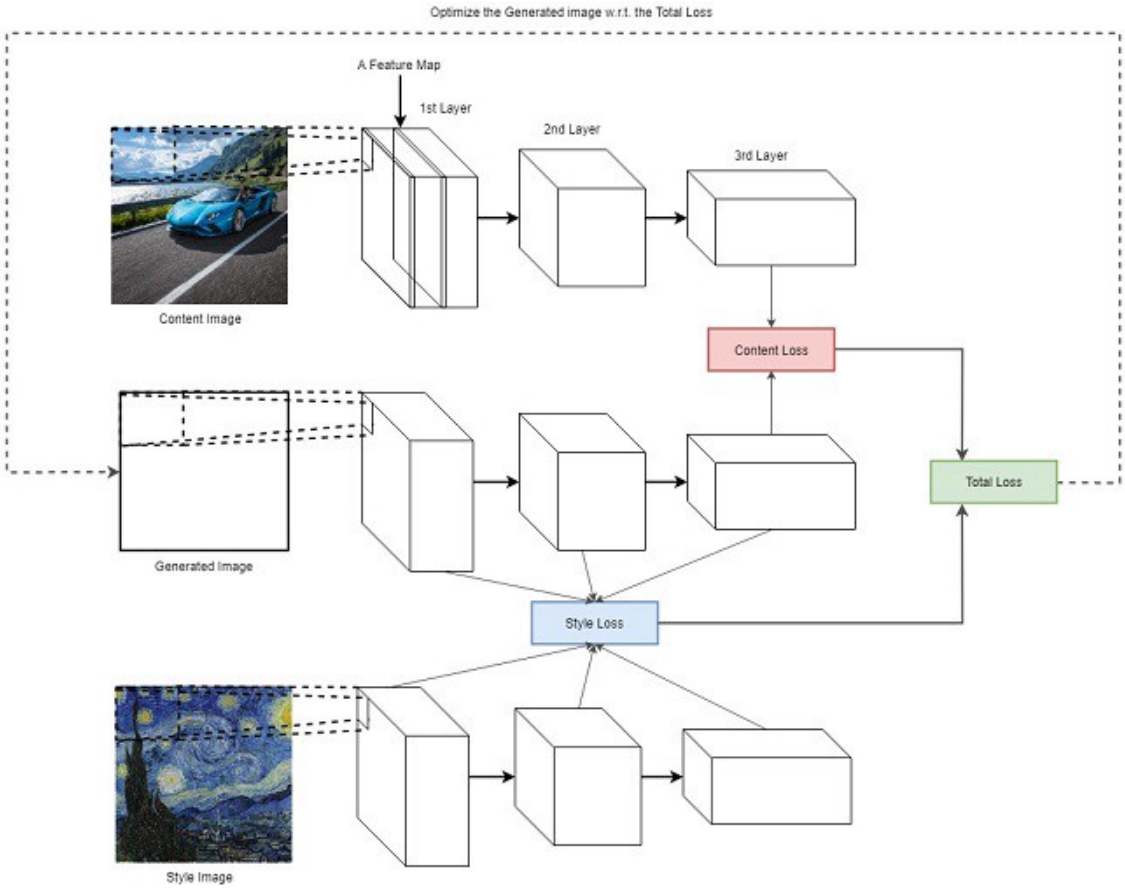
More works

- Aligned Structured Sparsity Learning for Efficient Image Super-Resolution
- ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks
- Learned Image Downscaling for Upscaling using Content Adaptive Resampler

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Style Transfer and Style Manipulation

A Neural Algorithm of Artistic Style



When CNNs are trained on object recognition, they develop a representation of the image that makes object information increasingly explicit along the processing hierarchy. Therefore, along the processing hierarchy of the network, the input image is transformed into representations that increasingly care about the actual content of the image compared to its detailed pixel values. We, therefore, refer to the feature responses in higher layers of the network as the content representation.

The key finding of this paper is that the representations of content and style in the Convolutional Neural Network are separable. That is, we can manipulate both representations independently to produce new, perceptually meaningful images.

Method: They used the feature space provided by the 16 convolutional and 5 pooling layers of the 19 layer VGG network. Generally, each layer in the network defines a non-linear filter bank whose complexity increases with the position of the layer in the network.

Hence a given input image x is encoded in each layer of the CNN by the filter responses to that image. A layer with N_l distinct filters has N_l feature maps each of size M_l , where M_l is the height times the width of the feature

map.

So the responses in a layer l can be stored in a matrix $F^l \in R^{N_l \times M_l}$ where $F_{ij}^{\ l}$ is the activation of the i-th filter at position j in layer l.

- **Content loss function -** the content loss function ensures that the activations of the higher layers are similar between the content image and the generated image. Let p and x be the original image and the generated image, and P^l and F^l their respective feature representation in layer l. We then define the squared-error loss between the two feature representations:

$$\mathcal{L}_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} \left(F_{ij}^l - P_{ij}^l\right)^2 .$$

since that CNN captures information about the content in the higher levels, the layer l is usually the top-most CNN layer.

The intuition behind the content loss - as we discussed previously about object localization using global average pooling (**"Learning Deep Features for Discriminative Localization"),** the feature maps in higher layers are activated in the presence of different objects. So if two images have the same content, they should have similar activations in the higher layers.

- **Style loss function -** the style loss function makes sure that the correlation of activations in all layers is similar between the style image and the generated image. To extract the style information from the VGG network, we use all the layers of the CNN (as opposed to using only the last layer in the content loss).

The style information is measured as the amount of correlation present between features maps in a given layer.

These feature correlations are given by the Gram matrix $G^l \in R^{N_l \times N_l}$, where $G_{ij}^{\ l}$ is the inner product between the vectorized feature map i and j in layer l:

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l.$$

To generate a texture that matches the style of a given image, we use gradient descent from a white noise image to find another image that matches the style representation of the original image. This is done by minimizing the mean-squared distance between the entries of the Gram matrix from the original image and the Gram matrix of the image to be generated. So let a and x be the original image and the image that is generated and A^l and G^l their respective style representations in layer l. The contribution of that layer to the total loss is then:

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^l - A_{ij}^l\right)^2$$

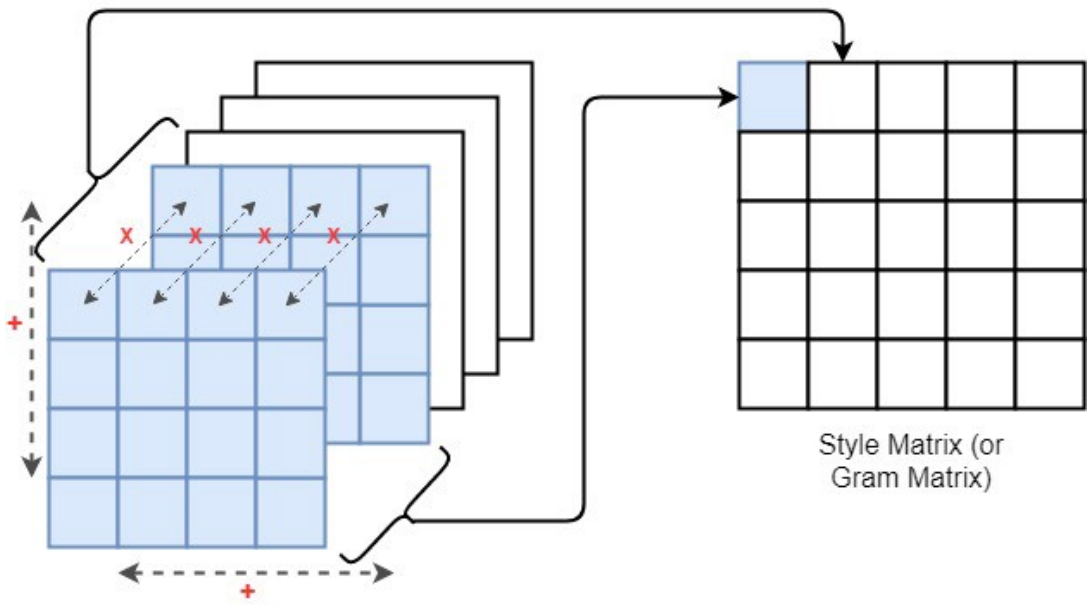
and the total loss is:

$$\mathcal{L}_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l$$

where w_l is a weight given to each layer during the loss computation.

The intuition behind the style loss - The goal is to compute a style matrix (visualized below) for the generated image and the style image. Then the style loss is defined as the root mean square difference between the two style matrices.

Below you can see an illustration of how the style matrix is computed. The style matrix is essentially a Gram matrix, where the (i,j) th element of the style matrix is computed by computing the element-wise multiplication of the i th and j th feature maps and summing across both width and height. In the figure, the red cross denotes element-wise multiplication and the red plus sign denotes summing across both width height of the feature maps.
The Gram matrix essentially captures the "distribution of features" of a set of feature maps in a given layer. By trying to minimize the style loss between two images, you are essentially matching the distribution of features between the two images



Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

StyleGAN

StyleGAN generates the artificial image gradually, starting from a very low resolution and continuing to a high resolution (1024×1024). By modifying the input of each level separately, it controls the visual features that are expressed in that level, from coarse features (pose, face shape) to fine details (hair color), without affecting other levels.

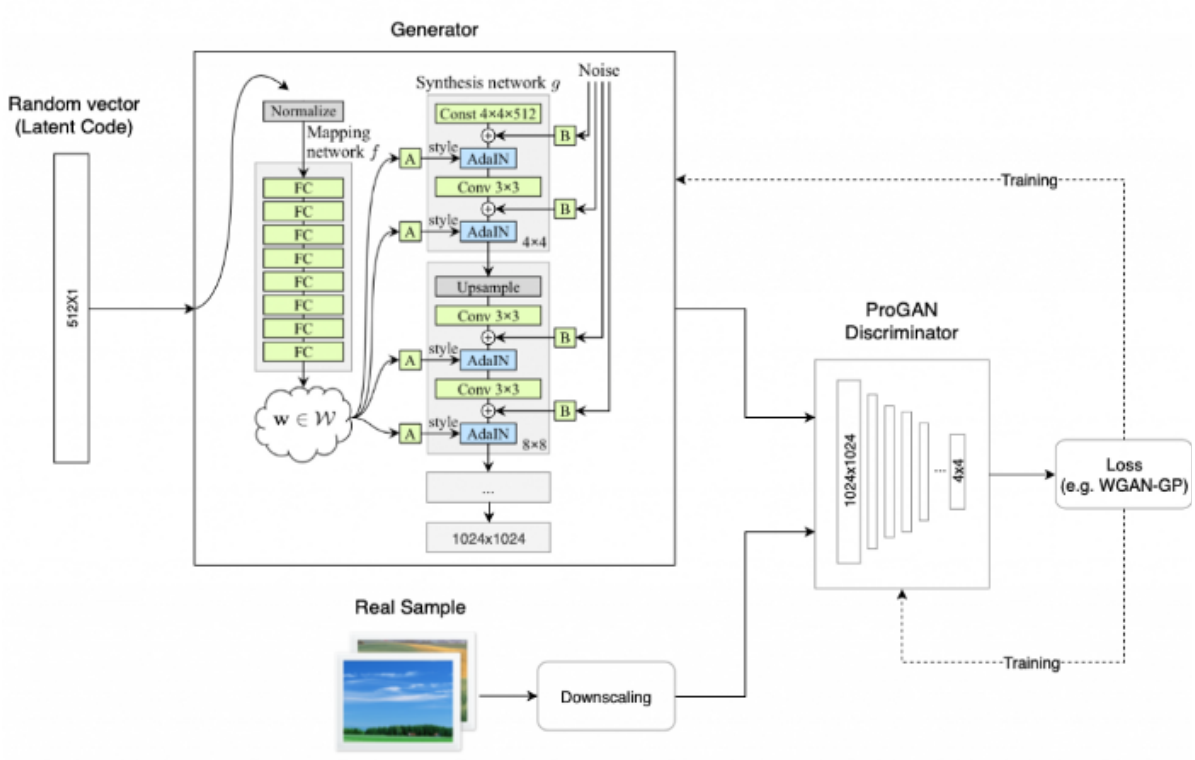
The StyleGAN paper offers an upgraded version of ProGAN's image generator, with a focus on the generator network. The authors observe that a potential benefit of the ProGAN progressive layers is their ability to control different visual features of the image if utilized properly. The lower the layer (and the resolution), the coarser the features it affects.

The paper divides the features into three types:

Coarse-resolution of up to 82 - effects pose, general hairstyle, face shape, etc

Middle - resolution of 162 to 322 - affects finer facial features, hairstyle, eyes open/closed, etc.

Fine - resolution of 642 to 10242 - affects color scheme (eye, hair, and skin) and micro features.

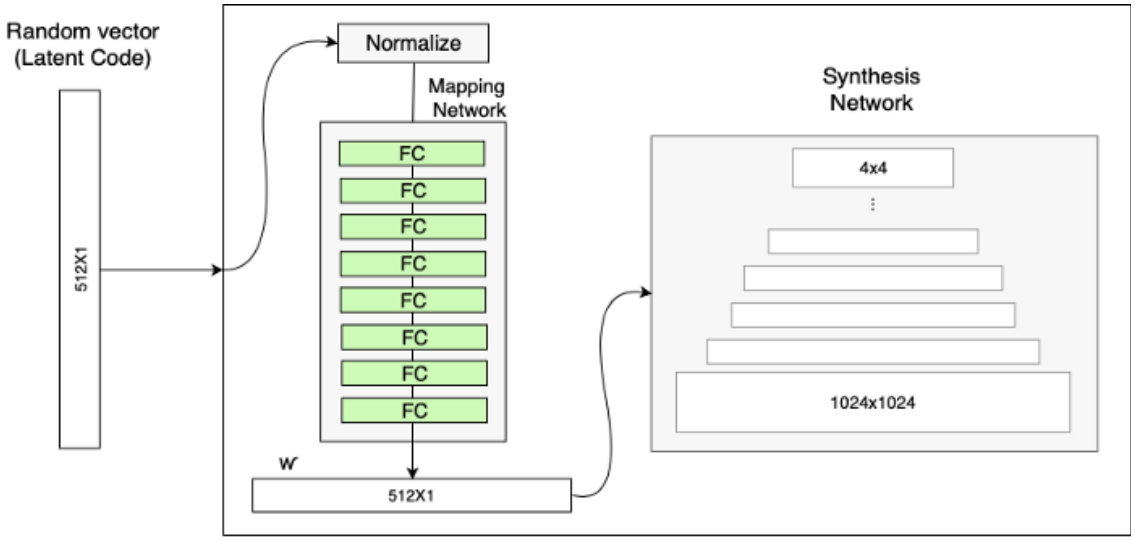


The Generator Architecture:

1) **mapping network (from latent vector to style vector)** - The Mapping Network's goal is to encode the input vector into an intermediate vector whose different elements control different visual features. This is a non-trivial process since the ability to control visual features with the input vector is limited, as it must follow the probability density of the training data. For example, if images of people with black hair are more common in the

dataset, then more input values will be mapped to that feature. As a result, the model isn't capable of mapping parts of the input (elements in the vector) to features, a phenomenon called features entanglement.

However, by using another neural network (i.e. mapping network) the model can generate a vector that doesn't have to follow the training data distribution and can reduce the correlation between features.
The Mapping Network consists of 8 fully connected layers and its output **w** is of the same size as the input layer (512×1). This network takes a randomly sampled point from the latent space as input and generates a style vector.



The style vector *w* is then transformed and incorporated into each block of the generator model after the convolutional layers via an operation called adaptive instance normalization or AdaIN.

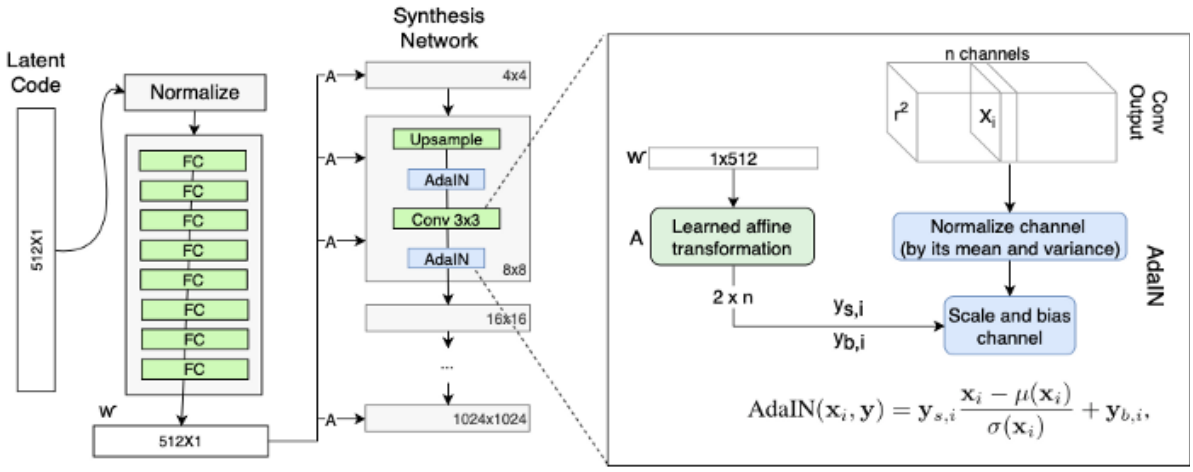
2) **Style Modules - AdaIN (Adding style to the generated images) -** The AdaIN (Adaptive Instance Normalization) module transfers the encoded information **w** created by the Mapping Network, into the generated image. The module is added to each resolution level of the Synthesis Network and defines the visual expression of the features in that level. In each resolution the AdaIN network does:

- Each channel of the convolution layer output is first normalized to make sure the scaling and shifting of step 3 have the expected effect.
- The intermediate vector w is transformed using another fully-connected layer (marked as A) into styles: $y = (y_s, y_b)$ where y_s is the scale and y_b is the bias for each channel.

- The scale and bias vectors shift each channel of the convolution output, thereby defining the importance of each filter in the convolution. In other words - The AdaIN layers involve first standardizing the output of the feature map to a standard Gaussian, then adding the style vector as a bias term.

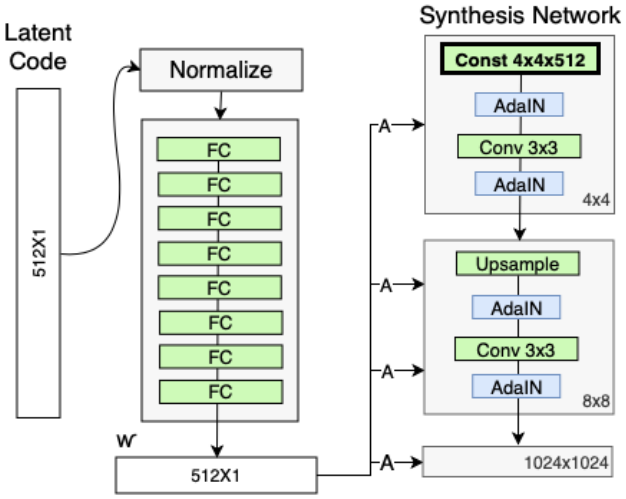
as can be seen in the image below, each feature map x_i is normalized separately, and then scaled and biased using the corresponding scalar components from style y.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



3) Removing the noise input

Most models use random noise as input to the generator. The StyleGAN team found that the image features are controlled by w and the AdaIN, and therefore the initial input can be omitted and replaced by constant values.

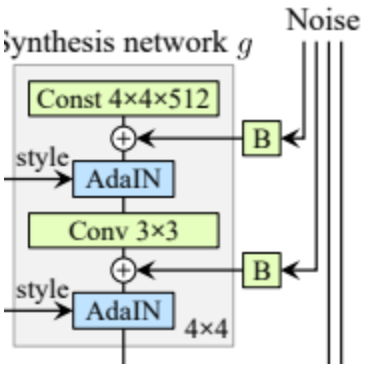


4) Stochastic variation (increase the variety of its outputs)-

There are many aspects in people's faces that are small and can be seen as stochastic, such as freckles, the exact placement of hairs, wrinkles, features that make the image more realistic and increase the variety of outputs. The common method to insert these small features into GAN images is adding random noise to the input vector. However, in many cases, it's tricky to control the noise effect due to the features entanglement phenomenon, which leads to other features of the image being affected.

The noise in StyleGAN is added in a similar way to the AdaIN mechanism — A scaled noise is added to each channel before the AdaIN module and changes a bit the visual expression of the features of the resolution level it operates on.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



5) Style mixing (to ensure better disentanglement of features at different levels) -

To further encourage the styles to localize, we employ mixing regularization using training, where a given percentage of images are generated using two random latent codes instead of one. Specifically, rhey run two latent codes It takes two inputs, generates the feature mapping vectors for each, then starts training using the first feature vector, and switches to the second one at a random level. This ensures the network will not rely on the correlation between levels. To be specific, we run two latent codes z1, z2 through the mapping network, and have the corresponding w1, w2 control the styles so that w1 applies before the crossover point and w2 after it. This regularization technique prevents the network from assuming that adjacent styles are correlated.

The use of different style vectors at different points of the synthesis network gives control over the styles of the resulting image at different levels of detail.

For example, blocks of layers in the synthesis network at lower resolutions (e.g. 4×4 and 8×8) control high-level styles such as pose and hairstyle. Blocks of layers in the model of the network (e.g. as 16×16 and 32×32) control hairstyles and facial expression. Finally, blocks of layers closer to the output end of the network (e.g. 64×64 to 1024×1024) control color schemes and very fine details.

Disentanglement studies

There are various definitions for disentanglement, but a common goal is a latent space that consists of linear subspaces, each of which controls one factor of variation. However, the sampling probability of each combination of factors in Z needs to match the corresponding density in the training data.

A major benefit of our generator architecture is that the intermediate latent space W does not have to support sampling according to any fixed distribution; its sampling density is induced by the learned piecewise continuous mapping f(z). This mapping can be adapted to “unwarp” W so that the factors of variation become more linear.

If a latent space is sufficiently disentangled, it should be possible to find direction vectors that consistently correspond to individual factors of variation. We propose a metric that quantifies this effect by measuring how well the latent-space points can be separated into two distinct sets via a linear hyperplane so that each set corresponds to a specific binary attribute of the image.

In order to label the generated images, we train auxiliary classification networks for a number of binary attributes, e.g., to distinguish male and female faces. In our tests, the classifiers had the same architecture as the discriminator we use and were trained using the CELEBA-HQ dataset that retains the 40 attributes available in the original CelebA dataset.

To measure the separability of one attribute, we generate 200,000 images with z ~ P(z) and classify them using the auxiliary classification network. We then sort the samples according to classifier confidence and remove the least confident half, yielding 100,000 labeled latent-space vectors.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

For each attribute, we fit a linear SVM to predict the label based on the latent-space point —z for traditional and w for style-based — and classify the points by this plane. We then compute the conditional entropy H(Y |X) where X are the classes predicted by the SVM and Y are the classes determined by the pre-trained classifier. This tells how much additional information is required to determine the true class of a sample, given that we know on which side of the hyperplane it lies. A low value suggests consistent latent space directions for the corresponding factor(s) of variation.

- **MUNIT**

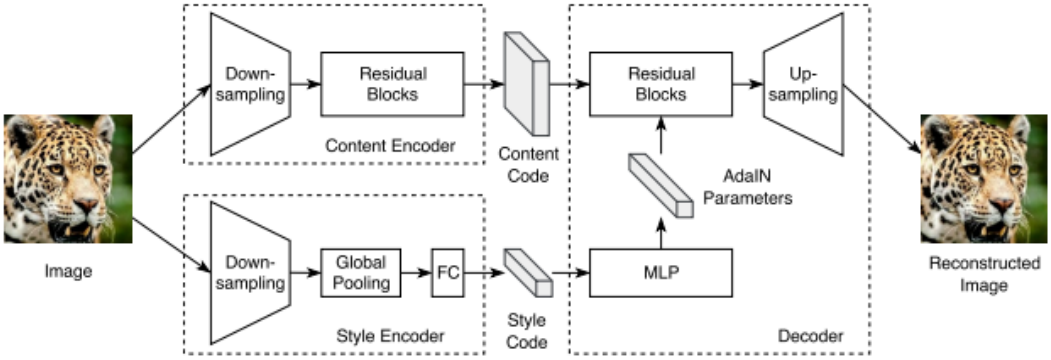


Fig. 3. Our auto-encoder architecture. The content encoder consists of several strided convolutional layers followed by residual blocks. The style encoder contains several strided convolutional layers followed by a global average pooling layer and a fully connected layer. The decoder uses a MLP to produce a set of AdaIN [54] parameters from the style code. The content code is then processed by residual blocks with AdaIN layers, and finally decoded to the image space by upsampling and convolutional layers.

Style-GAN's latent space (background)

Most of the research on StyleGAN's latent space revolves around two separate tasks: GAN inversion and latent space manipulation.

1. **Inversion** - given an image x, we infer a latent code w, which is used to reconstruct x as accurately as possible when forwarded through the generator G.

2. **Latent space manipulation** - for a given latent code w, we infer a new latent code, w' , such that the synthesized image G(w') portrays a semantically meaningful edit of G(w). Inspecting the GAN inversion task, we stress that reconstruction alone has a limited purpose since reconstructing an inverted image can at best return the original image.

The central motivation for inversion is to allow for further latent editing operations. That is, a successful reconstruction should enable extensive and diverse manipulation of real images with greater ease. This objective, however, is not trivial as some latent codes are more editable than others.

There are two separate properties of reconstruction — distortion and perceptual quality. Formally, distortion is defined as Ex~pX [△(x, G(w)] where pX is the distribution of the real images, and △(x, G(w)) is an image-space

difference measure between images x and G(w). Perceptual quality measures how realistic the reconstructed images are, with no relation to any reference image.

Most previous works have considered only distortion for evaluating the reconstruction. Distortion alone, however, does not capture the quality of the reconstruction.

Image2StyleGAN: How to Embed Images Into the StyleGAN Latent Space?

Propose an efficient algorithm to embed a given image into the latent space of StyleGAN. This embedding enables semantic image editing operations that can be applied to existing photographs.

There are multiple latent spaces in StyleGAN that could be used for embedding. Two obvious candidates are the initial latent space Z and the intermediate latent space W. The 512-dimensional vectors w ∈ W are obtained from the 512-dimensional vectors z ∈ Z by passing them through a fully connected neural network. An important insight of our work is that it is not easily possible to embed into W or Z directly. Therefore, we propose to embed into an extended latent space W+. W+ is a concatenation of 18 different 512-dimensional w vectors, one for each layer of the StyleGAN architecture that can receive input via AdaIn.

(notice that styleGAN generates one $w \in W$ vector using the mapping network, and then broadcasts this vector to all 18 layers of the generator synthesis. In each layer, the w vector is transformed to y vector using affine transformation - fully connected layer. the w vector contains global style information regarding the image, and it is mapped using the affine transformation to 18 style vectors that each one of them contains different information - fine, medium, and coarse information).

Embedding Algorithm

Our method follows a straightforward optimization framework to embed a given image onto the manifold of the pre-trained generator. Starting from a suitable initialization w, we search for an optimized vector w* that minimizes the loss function that measures the similarity between the given image and the image generated from w* . To measure the similarity between the input image and the embedded image during optimization, we employ a loss function that is a weighted combination of the VGG-16 perceptual loss and the pixel-wise MSE loss

Algorithm 1: Latent Space Embedding for GANs

Input: An image $I \in \mathbb{R}^{n \times m \times 3}$ to embed; a pre-trained generator $G(\cdot)$.

Output: The embedded latent code w^* and the embedded image $G(w^*)$ optimzed via F'.

Initialize latent code $w^* = w$;

while *not converged* **do**

  $L \leftarrow L_{percept}(G(w^*), I) + \frac{\lambda}{N}\|G(w^*) - I\|_2^2$;

  $w^* \leftarrow w^* - \eta F'(\nabla_{w^*} L)$;

end

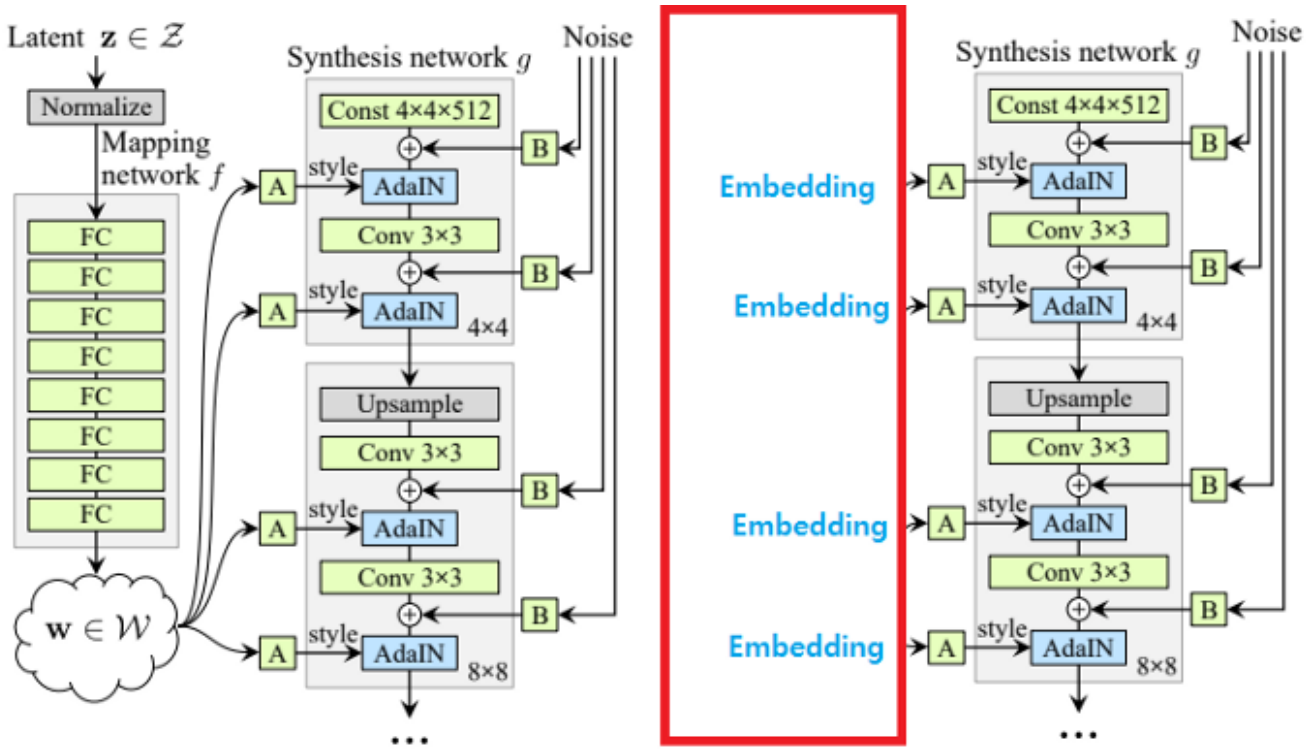
1. embedding to Z:
   - latents = torch.zeros(1, 512, requires_grad=True, device=device)

- for epoch in epochs:
    - w = g_mapping(latents)
    - w1= w.unsqueeze(1).expand(-1, 18, -1)
    - synt_image = g_synthesis(w1)
    - loss = perceptual_loss + MSE
    - loss.backword(), optimizer.step()

2. embedding to W:
    - latent_w = torch.zeros(1, 512, requires_grad=True, device=device)
    - for epoch in epochs:
        - w1= latent_w .unsqueeze(1).expand(-1, 18, -1)
        - synt_image = g_synthesis(w1)
        - loss = perceptual_loss + MSE
        - loss.backword(), optimizer.step()

3. embedding to W+:

- latent_w = torch.zeros(1, 18, 512, requires_grad=True, device=device)
- for epoch in epochs:
    - synt_image = g_synthesis(latent_w)
    - loss = perceptual_loss + MSE
    - loss.backword(), optimizer.step()



Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Encoding in Style: a StyleGAN Encoder for Image-to-Image Translation

StyleGAN proposes a novel style based generator architecture and attains state-of-the-art visual quality on high-resolution images. Moreover, it has been demonstrated that it has a disentangled latent space, W, which offers control and editing capabilities.
Recently, numerous methods have shown competence in controlling StyleGAN's latent space and performing meaningful manipulations in W. These methods follow an "invert first, edit later" approach, where one first inverts an image into StyleGAN's latent space and then edits the latent code in a semantically meaningful manner to obtain a new code that is then used by StyleGAN to generate the output image. However, it has been shown that inverting a real image into a 512-dimensional vector $w \in W$ does not lead to an accurate reconstruction. Motivated by this, it has become common practice to encode real images into an extended latent space, W+, defined by the concatenation of 18 different 512-dimensional w vectors, one for each input layer of StyleGAN. These works usually resort to using per-image optimization over W+, requiring several minutes for a single image. To accelerate this optimization process, some methods have trained an encoder to infer an approximate vector in W+ which serves as a good initial point from which additional optimization is required. However, a fast and accurate inversion of real images into W+ remains a challenge.

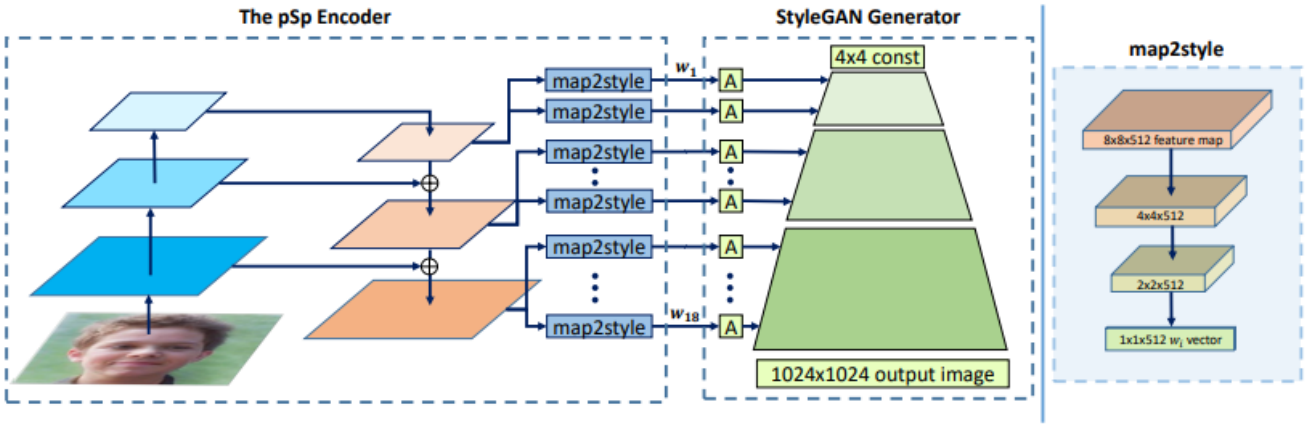


Figure 2. Our pSp architecture. Feature maps are first extracted using a standard feature pyramid over a ResNet backbone. For each of the 18 target styles, a small mapping network is trained to extract the learned styles from the corresponding feature map, where styles (0-2) are generated from the small feature map, (3-6) from the medium feature map, and (7-18) from the largest feature map. The mapping network, *map2style*, is a small fully convolutional network, which gradually reduces spatial size using a set of 2-strided convolutions followed by LeakyReLU activations. Each generated 512 vector, is fed into StyleGAN, starting from its matching affine transformation, A.

Our pSp framework builds upon the representative power of a pretrained StyleGAN generator and the W+ latent space. In StyleGAN, the authors have shown that the different style inputs correspond to different levels of detail, which are roughly divided into three groups — coarse, medium, and fine.

Following this observation, in pSp we extend an encoder backbone with a feature pyramid, generating three levels of feature maps from which styles are extracted using a simple intermediate network — map2style. The styles, aligned with the hierarchical representation, are then fed into the generator in correspondence to their scale to generate the output image, thus completing the translation from input pixels to output pixels, through the intermediate style representation

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

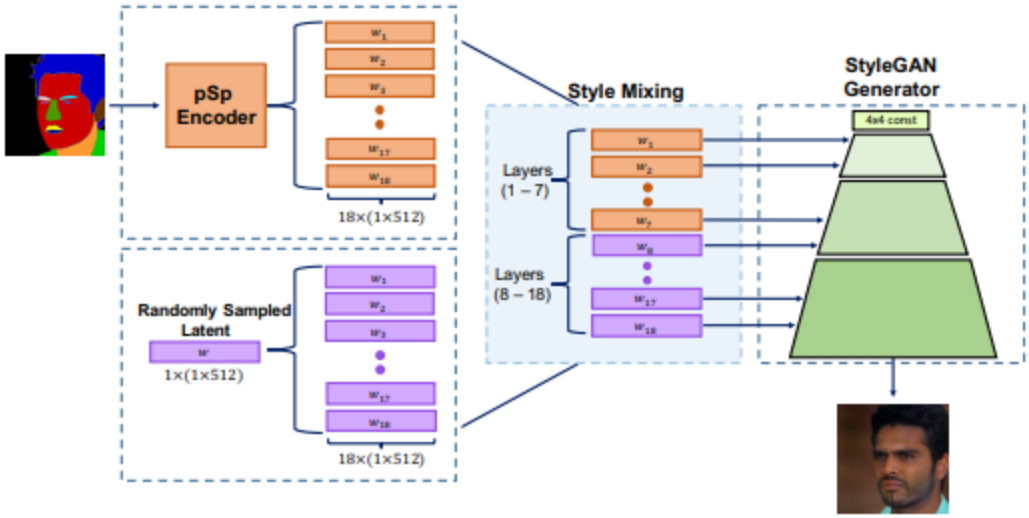


Figure 3. Style-mixing for multi-modal generation.

Designing an Encoder for StyleGAN Image Manipulation

To perform manipulations on real images, one must first obtain the latent code from which the pretrained GAN can most accurately reconstruct the original input image. This task has been commonly referred to as GAN Inversion. Generally, inversion methods either (i) directly optimize the latent vector to minimize the error for the given image, (ii) train an encoder to map the given image to the latent space, or (iii) use a hybrid approach combining both. Typically, methods performing optimization are superior in achieving low distortion. However, they require a substantially longer time to invert an image and are less editable.

Terminology

StyleGAN consists of two key components: (i) a mapping function that maps a latent code $z \in Z = N(\mu, \sigma^2)$ into a style code $w \in W \subseteq R^{512}$, and (ii) a generator that takes in the style code, replicated several times (according to the desired resolution), and generates an image.

Note that while the distribution Z is known to be Gaussian, there is no known explicit model for the distribution of W. In the following, we will refer to this distribution as the range of the mapping function. It has been shown that not every real, in-domain image can be inverted into StyleGAN's latent space. To alleviate this limitation, one can increase the expressiveness of the StyleGAN generator by inputting $k$different style codes instead of a single vector. We denote this extended space by $W^k \subseteq R^{k \times 512}$where k is the number of style inputs of the generator. For example, a generator capable of synthesizing images at a resolution of 1024 × 1024 operates in the extended W^{18} space corresponding to the 18 different style inputs.

Even more expressive power can be achieved by inputting style codes that are not necessarily from the true distribution of W, i.e. outside the range of StyleGAN's mapping function. Observe that this extension can be applied by taking a single style code and replicating it, or by taking k different style codes. We denote these

extensions by W^* and W^{k}_{*}, respectively. Note, that here, we depart from the commonly used W+ notation due to its ambiguity with various works referring to it as both W^k and W^{k}_{*}.

However, note, that for simplicity and convenience, we refer to W both as the 512-dimensional distribution, and as a subset of W_k where all the k style codes are equal and in$W$.

W^{k}_{*} differs from Win two ways. First, W^{k}_{*}may contain different style codes at different style-modulation layers. Second, each style code, independently, is not bound to the true distribution of W, but can instead take any value fromR^{512} . It is well known that W^{k}_{*} achieves lower, i.e. better, distortion than$W$. Additionally, we find that W is more editable.

Figure 4: The editability gap between $\mathcal{W}$ and $\mathcal{W}^k_*$. The top row depicts several edits performed on an image generated from a latent code sampled from $\mathcal{W}$. In the bottom row, the same image is inverted back into $\mathcal{W}^k_*$ and passed through the generator to obtain a visually-similar image. The same edits as those performed in the top row are then applied on the inverted $\mathcal{W}^k_*$ code. Note that although the two source images are similar in the image-space, the edits in $\mathcal{W}$ are visually better than the corresponding ones in $\mathcal{W}^k_*$, as can be observed by the warped shapes of the cars in the bottom row.

Figure 5: An example of the distortion-perception tradeoff. Inverting the source image using two different encoders yields significantly different results. While the middle image achieves low distortion (notice the overall similarity in shape and posture) and low perceptual quality (notice the warped head), the rightmost image achieves higher distortion but better perceptual quality.

This provides the first evidence of the inherent tradeoff between distortion and editability. Observe that since StyleGAN is originally trained in the W space, it is not surprising that W is more well-behaved and has better perceptual quality compared to its W^{k}_{*} counterpart. On the other hand, observe that due to the significantly higher dimensionality of W^{k}_{*} and the architecture of StyleGAN,W^{k}_{*} has far greater expressive power

Designing an encoder

we now present principles for designing an encoder and novel training scheme to explicitly address the proximity to W. In the following, we consider a generic encoder that infers latent codes in the space of W^{k}_{*} . We now present the two principles for controlling the proximity to W, where each is defined using a dedicated training paradigm to encourage the encoder to map into regions in W^{k}_{*} that lie close to W.

1. minimizing variation - The first approach for getting closer to W is to encourage the inferredW^{k}_{*} latent codes to lie closer to W∗, i.e. minimize the variance between the different style codes, or equivalently,

encourage the style codes to be identical. Let E(x) = (w0, w1, ..., wN−1) denote the output of the encoder, where N is the number of style-modulation layers. Common encoders are trained directly into $W^k{}_*$, i.e., learn each wi separately and simultaneously. Conversely, we infer a single latent code, w, and a set of offsets from w. More formally, we learn an output of the form E(x) = (w, w + Δ1, ..., w + ΔN−1). Specifically, at the start of training, we set ∀ i : △i = 0 and the encoder is trained to infer a single W∗ code. We then gradually allow the network to learn different Δi for each i sequentially. We find that doing so is useful in allowing the encoder to gradually expand from W∗ towards $W^k{}_*$. This scheme, together with the semantic meaning of specific input layers of StyleGAN, allows one to first learn a coarse reconstruction of the input image and then gradually add finer details while still remaining close to W∗. We note that low-frequency details greatly control the distortion quality. Thus, our progressive training scheme first focuses on improving the low frequency distortion by tuning the coarse-level offsets. Then, the encoder gradually complements these offsets with higher frequency details introduced by the finer-level offsets. In order to explicitly enforce a proximity to W∗, we add an L2 delta-regularization loss:

$$\mathcal{L}_{\text{d-reg}}(w) = \sum_{i=1}^{N-1} ||\Delta_i||_2.$$

2. Minimize Deviation From W^k - The second approach for getting closer to W is to encourage the $W^k{}_*$ latent codes obtained by the encoder to lie closer toW^k . That is, encourage the individual style codes to lie within the actual distribution of W. To do so, we adopt a latent discriminator which is trained in an adversarial manner to discriminate between real samples from the W space (generated by StyleGAN's mapping function) and the encoder's learned latent codes. We use a single latent discriminator, denoted DW, that operates on each latent code entry separately. In every iteration, we calculate the GAN loss for every E(x)i (fake) and Dw(w) (real latent) and average over all i-s.

$$\mathcal{L}_{\text{adv}}^{D} = -\mathop{\mathbb{E}}_{w\sim\mathcal{W}}[\log D_{\mathcal{W}}(w)] - \mathop{\mathbb{E}}_{x\sim p_X}[\log(1 - D_{\mathcal{W}}(E(x)_i)] + \frac{\gamma}{2}\mathop{\mathbb{E}}_{w\sim\mathcal{W}}\left[\|\nabla_w D_{\mathcal{W}}(w)\|_2^2\right], \quad (2)$$

$$\mathcal{L}_{\text{adv}}^{E} = -\mathop{\mathbb{E}}_{x\sim p_X}[\log D_{\mathcal{W}}(E(x)_i)]. \quad (3)$$

e4e: encoder for editing

Unlike the original pSp encoder which generates N style codes in parallel, here, we generate a single base style code, denoted by w, and a series of N − 1 offset vectors. The offsets are then summed up with the base style code w to yield the final N style codes which are then fed into a fixed, pretrained StyleGAN2 generator to obtain the reconstructed image

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

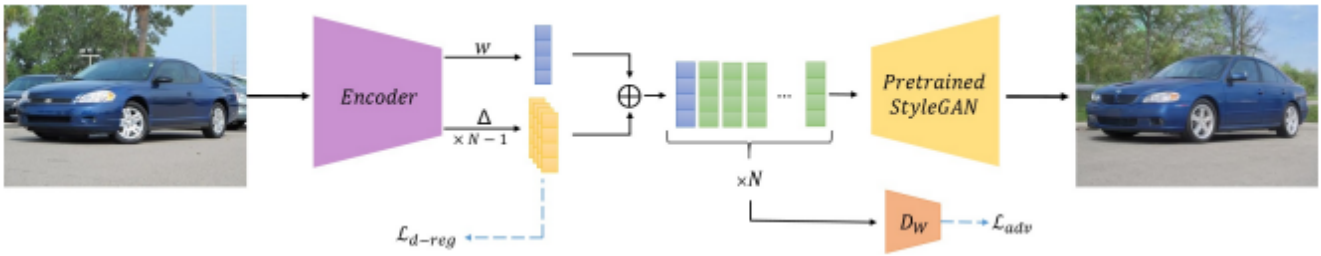


Figure 6: Our e4e network architecture. The encoder receives an input image and outputs a single style code w together with a set of offsets $\Delta_1..\Delta_{N-1}$, where N denotes the number of StyleGAN's style modulation layers. We obtain our final latent representation by replicating the w vector N times and adding each Δ_i to its corresponding entry. During training, the $\mathcal{L}_{\text{d-reg}}$ regularization encourages small variance between different entries of the final representation, thereby remaining close to $\mathcal{W}_*$. $\mathcal{L}_{\text{adv}}$ guides each latent code towards the range of StyleGAN's mapping network, resulting in a final representation closer to $\mathcal{W}^k$. As a result of applying both regularization terms, the encoder's final learned representation lies close to $\mathcal{W}$.

Losses:

1) distortion
   - cosine similarity - minimize the cosine similarity between the feature embeddings (apply resnet on the image) of the reconstructed image and its source image.
   - L2 loss
   - perceptual loss
2) perceptual quality and editability - First, we apply a delta-regularization loss to ensure proximity to W* when learning the offsets Δi . Second, we use an adversarial loss using our latent discriminator, which encourages each learned style code to lie within the distribution W.

StyleCLIP - text guided image manipulation

Inspired by the ability of StyleGAN to generate highly realistic images in a variety of domains, much recent work has focused on understanding how to use the latent spaces of StyleGAN to manipulate generated and real images. However, discovering semantically meaningful latent manipulations typically involves a painstaking human examination of the many degrees of freedom or an annotated collection of images for each desired manipulation. In this work, we explore leveraging the power of recently introduced Contrastive Language-Image Pre-training (CLIP) models in order to develop a text-based interface for StyleGAN image manipulation that does not require such manual effort.

CLIP learns a multi-modal embedding space, which may be used to estimate the semantic similarity between a given text and an image.

In this work we explore three ways for text-driven image manipulation, all of which combine the generative power of StyleGAN with the rich joint vision-language representation learned by CLIP:

1) simple latent optimization scheme, where a given latent code of an image in StyleGAN's W+ space is optimized by minimizing a loss computed in CLIP space. Specifically, given a source latent code ws ∈ W+, and a directive in natural language, or a text prompt t, we solve the following optimization problem:

$$\arg\min_{w \in \mathcal{W}+} D_{\text{CLIP}}(G(w), t) + \lambda_{\text{L2}} \|w - w_s\|_2 + \lambda_{\text{ID}} \mathcal{L}_{\text{ID}}(w),$$

where G is a pretrained StyleGAN generator and DCLIP is the cosine distance between the CLIP embeddings of its two arguments. Similarity to the input image is controlled by the L2 distance in latent space, and by the identity loss:

$$\mathcal{L}_{\text{ID}}(w) = 1 - \langle R(G(w_s)), R(G(w)) \rangle,$$

where R is a pretrained ArcFace network for face recognition, and <,> computes the cosine similarity between its arguments.

2) mapping network is trained to infer a manipulation step in latent space, in a single forward pass. The training takes a few hours, but it must only be done once per text prompt. The direction of the manipulation step may vary depending on the starting position in W+, which corresponds to the input image, and thus we refer to this mapper as local.

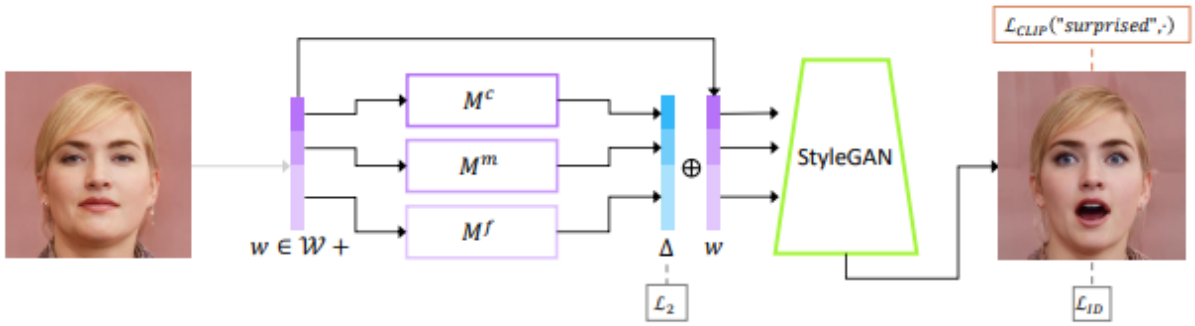


Figure 2. The architecture of our text-guided mapper (using the text prompt "surprised", in this example). The source image (left) is inverted into a latent code w. Three separate mapping functions are trained to generate residuals (in blue) that are added to w to yield the target code, from which a pretrained StyleGAN (in green) generates an image (right), assessed by the CLIP and identity losses.

It has been shown that different StyleGAN layers are responsible for different levels of detail in the generated image. Consequently, it is common to split the layers into three groups (coarse, medium, and fine), and feed each group with a different part of the (extended) latent vector. We design our mapper accordingly, with three fully-connected networks, one for each group/part. The architecture of each of these networks is the same as that of the StyleGAN mapping network, but with fewer layers (4 rather than 8, in our implementation). Denoting the latent code of the input image as $w = (w_c, w_m, w_f)$, the mapper is defined by:

$$M_t(w) = (M_t^c(w_c), M_t^m(w_m), M_t^f(w_f)).$$

Our mapper is trained to manipulate the desired attributes of the image as indicated by the text prompt t, while preserving the other visual attributes of the input image. The CLIP loss, $L_{CLIP}(w)$) guides the mapper to minimize the cosine distance in the CLIP latent space:

$$\mathcal{L}_{\text{CLIP}}(w) = D_{\text{CLIP}}(G(w + M_t(w)), t),$$

where G denotes again the pretrained StyleGAN generator. To preserve the visual attributes of the original input image, we minimize the L2 norm of the manipulation step in the latent space. Finally, for edits that require identity preservation, we use the identity loss. Our total loss function is a weighted combination of these losses:

$$\mathcal{L}(w) = \mathcal{L}_{\text{CLIP}}(w) + \lambda_{L2} \left\| M_t(w) \right\|_2 + \lambda_{\text{ID}} \mathcal{L}_{\text{ID}}(w).$$

Image inpainting

Image inpainting is the task of filling missing pixels in an image such that the completed image is realistic-looking and follows the original (true) context.
For image inpainting, texture details of the filled pixels are important. The valid pixels and the filled pixels should be consistent and the filled images should look realistic.

Additional applications:

- remove distracting objects
- retouch undesired regions in photos (example: if we want to remove glasses of a person, we can mask the glasses as the “missing region” we want to fill, and the model will fill this region with alternative content based on the context/previous images in the training data).

Two main loss components:

- pixel-wise reconstruction loss (i.e. L2 loss) to ensure that we can fill in the missing parts with “correct” structure.
- GAN loss (i.e. Adversarial loss) and/or texture loss should be used to obtain the filled images with sharper texture details of the generated pixels.

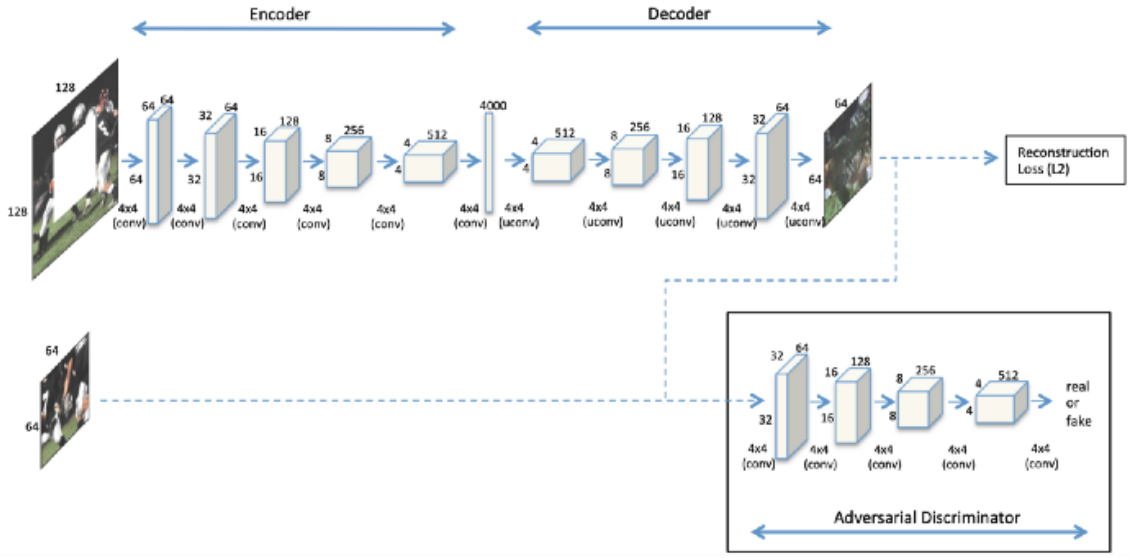
Pre deep learning: using image statistics of the remaining image to fill in the hole. PatchMatch, one of the state-of-the-art methods, iteratively searches for the best fitting patches to fill in the holes. While this approach generally produces smooth results, it is limited by the available image statistics and has no concept of visual semantics.

- What is the problem with patch-based methods?

  For patch-based methods, one heavy assumption is that we believe we can find similar patches outside the missing regions and these similar patches would be useful for filling in the missing regions. This assumption may be true for natural scenes as sky and lawn can have many similar patches in an image. What if there are not any similar patches outside the missing regions just like the case of face image inpainting. For such a case, we cannot find any eye patches to fill in the corresponding missing parts. Therefore, robust inpainting algorithms should be able to generate novel fragments.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Context Encoder (2016) - The first GAN based inpainting model.



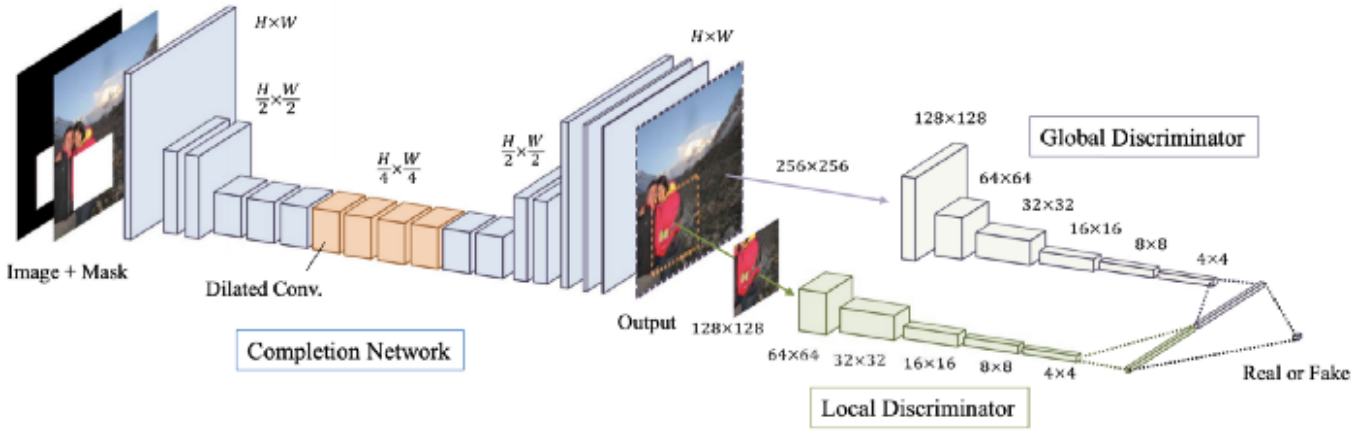
The term “context” relates to the understanding of the entire image itself. We call our model context encoder, as it consists of an encoder capturing the context of an image into a compact latent feature representation and a decoder that uses that representation to produce the missing image content.

Main idea: Deep semantic understanding of an image or the context of an image is important to the task of inpainting, and (channel-wise) fully-connected layer is one way to capture the context of an image.

- Loss: jointly training the context encoders to minimize both a reconstruction loss and an adversarial loss. The reconstruction (L2) loss captures the overall structure of the missing region in relation to the context, while the adversarial loss has the effect of picking a particular mode from the distribution. using only the reconstruction loss produces blurry results, whereas adding the adversarial loss results in much sharper predictions.
    - reconstruction loss - MSE between the completed region and the ground truth region
    - adversarial loss - the discriminator gets the ground truth image, and the completed image is generated by the context encoder and needs to distinguish between them. In other words - maximize D(x) and minimize $D(F((1 - M) \odot x)))$ where M is a binary mask with “1” on the hole that needs to be filled and 0 for valid pixels, and F is the context encoder network.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

GLCIC - Globally and Locally Consistent Image Completion (2017)



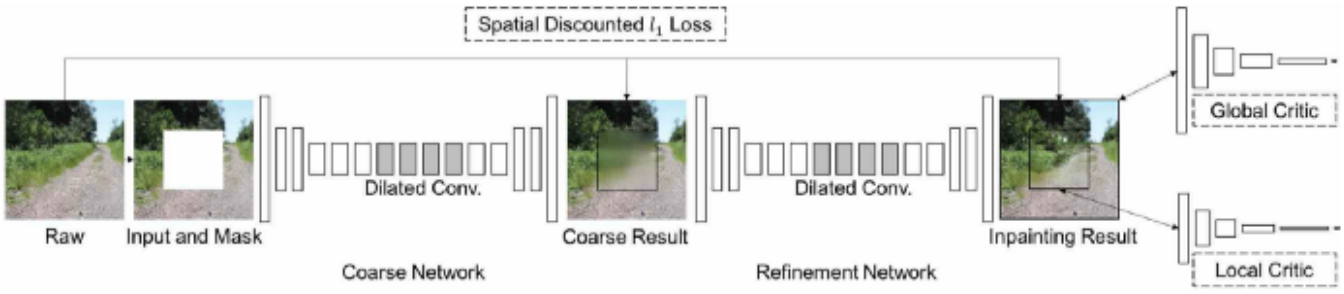
Globally and Locally Consistent Image Completion (GLCIC, 2017) is a milestone in deep image inpainting as it defines the Fully Convolution Network with Dilated Convolutions for deep image inpainting and actually this is a typical network structure for deep image inpainting.

- the architecture is composed of three networks:
    - A completion network - fully convolutional network that is used to complete the image. The input to this network is an RGB image with a binary channel that indicates the image completion mask (1 for a pixel to be completed) and the output is an RGB image. The general architecture follows an encoder decoder structure, which allows reducing the memory usage and computational time by initially decreasing the resolution before further processing the image. Afterwards, the output is restored to the original resolution using deconvolution layers, which consist of convolutional layers with fractional strides.
    - global context discriminator - takes the full image as input to recognize global consistency of the scene. It consists of six convolutional layers and a single fully-connected layer that outputs a single 1024- dimensional vector. All the convolutional layers employ a stride of 2 × 2 pixels to decrease the image resolution while increasing the number of output filters.
    - Local context discriminator - looks only at a small region around the completed area in order to judge the quality of more detailed appearance. The local context discriminator follows the same pattern as the global discriminator, except that the input is a 128 × 128-pixel image patch centered around the completed region.
    - Finally, the outputs of the global and the local discriminators are concatenated together into a single 2048-dimensional vector, which is then processed by a single fully-connected layer, to output a continuous value. A sigmoid transfer function is used so that this value is in the [0, 1] range and represents the probability that the image is real, rather than completed.
- **They use dilated convolution instead of the fully connected layer** - understanding about the context of the entire image is important to the task of image inpainting. Previous approaches employ a fully-connected layer as the middle layer so as to understand the context. Remember that the standard convolution layer performs convolution at local regions while the fully-connected layer fully connects all the neurons (i.e. each output value depends on all the input values). However, a fully-connected layer imposes limitation on the input image size and induces much more learnable parameters.

Dilated convolutions use kernels that are spread out, allowing to compute each output pixel with a much larger input area, while still using the same amount of parameters and computational power. This is important for the image completion task, as the context is critical for realism. By using dilated convolutions at lower resolutions, the model can effectively "see" a larger area of the input image when computing each output pixel than with standard convolutional layers.

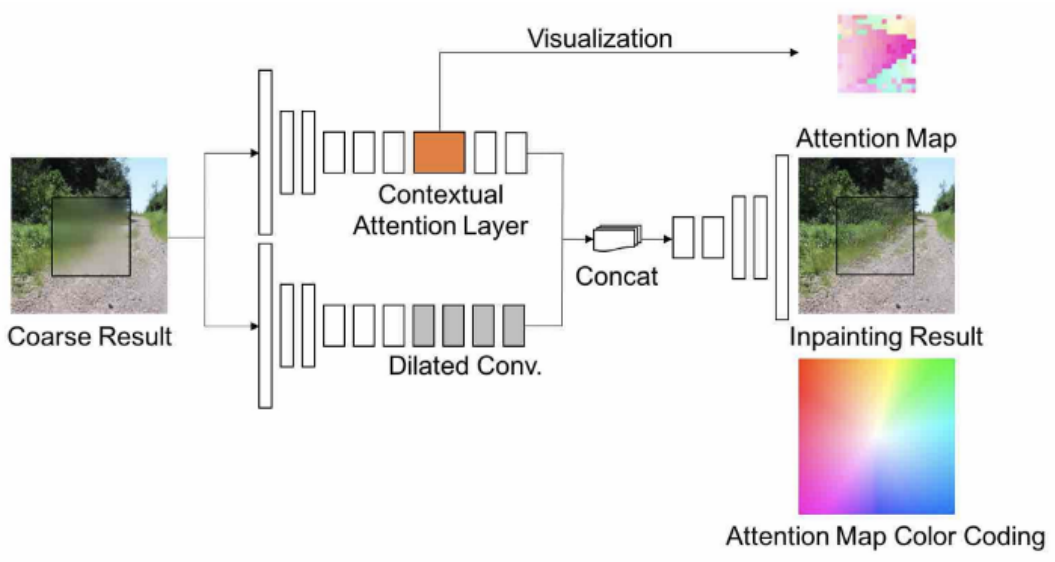
- Apart from the fully convolutional network with dilated convolutions, two discriminators at two scales were also trained together with the generator network. A global discriminator looks at the whole image while a local discriminator looks at the filled centre hole. With both the global and local discriminators, the filled image would have better global and local consistency. Note that many later inpainting papers follow this multi-scale discriminator design. To solve these limitations, dilated convolution is used to construct a fully convolutional network that allows input at various sizes. On the other hand, by adjusting the dilation rate of a standard kernel (usually 3×3), we can have larger receptive fields at different layers to help understanding the context of the entire image.
- training losses:
  let $C(x,\ M_c)$ denote the completion network, with x the input image and M_c the completion region mask that is the same size as the input image. The binary mask M_c takes the value 1 inside regions to be filled-in and 0 elsewhere.
  1) Weighted MSE loss between the completed hole and the groundtruth pixels in this hole.
  2) GAN loss - the completion network acts as the generator that tries to fool the discriminator. During the training, the context discriminators are trained to distinguish fake from real images, while the completion network is trained to deceive the discriminators.

DeepFill V1



The proposed framework consists of two generator networks and two discriminator networks. The two generators follow the fully convolutional networks with dilated convolutions. One generator is for coarse reconstruction and another one is for refinement. This is called standard coarse-to-fine network structure. The two discriminators also look at the completed images both globally and locally. The global discriminator takes the entire image as input while the local discriminator takes the filled region as input.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



Contextual Attention mechanism

Contextual Attention mechanism is proposed to effectively borrow the contextual information from distant spatial locations for reconstructing the missing pixels. The contextual attention is applied to the second refinement network. The first coarse reconstruction network is responsible for a rough estimation of the missing regions.

The contextual attention layer learns where to borrow or copy feature information from known background patches to generate missing patches.

We consider the problem where we want to match features of missing pixels (foreground) to surroundings (background). We first extract patches (3 × 3) in the background and reshape them as convolutional filters.

To match foreground patches {fx,y} with background ones {bx' ,y'}, for each foreground patch and background patch, we measure the similarity of patch centered in background (x', y') and foreground (x, y):

$$s_{x,y,x',y'} = \langle \frac{f_{x,y}}{||f_{x,y}||}, \frac{b_{x',y'}}{||b_{x',y'}||} \rangle,$$

The similarity is calculated using F.conv2D, where the input is the foreground patches(each patch of size 64x64), and the filters are background patches - each background patch of size 3x3 and there are 128 background patches overall. so the result of the conv2d is 128 feature maps, each feature map of size 64x64 and each feature map represents the similarity between one background patch and all the patches inside the missing region.

Then we weight the similarity with scaled softmax along x'y' - dimension to get attention score for each pixel - for each foreground patch, we get a score vector of size 128 for each background patch. The score represents the similarity of each background patch to this foreground patch.

Finally, we reconstruct the generated features inside the missing region by deconvolution using the attention feature maps as input and the known patches as kernels.

Image Inpainting for Irregular Holes Using Partial Convolutions (2018)

Motivation: deep generative models based on vanilla convolutions are naturally ill-fitted for image hole-filling because the spatially shared convolutional filters treat all input pixels or features the same as valid ones. For holefilling, the input to each layer are composed of valid pixels/features outside holes and invalid ones in masked regions. Vanilla convolutions apply the same filters on all valid, invalid and mixed (for example, the ones on hole boundary) pixels/features, leading to visual artifacts such as color discrepancy, blurriness and obvious edge responses surrounding holes when tested on free-form masks.

Solution: The key idea is to separate the missing pixels from the valid pixels during convolutions such that the results of convolutions only depend on the valid pixels. They propose the use of a Partial Convolutional Layer, comprising a masked and re-normalized convolution operation followed by a mask-update step.

The use of partial convolutions is such that given a binary mask the convolutional results depend only on the non-hole regions at every layer. The main extension is the automatic mask update step, which removes any masking where the partial convolution was able to operate on an unmasked value. Given sufficient layers of successive updates, even the largest masked holes will eventually shrink away, leaving only valid responses in the feature map.

- Partial Convolution layer

  given X, the pixel values for the current convolution (sliding) window, and M the binary mask, before applying the convolution kernel W on X as usual, we element-wise multiply the window by the mask so the output values depend only on the unmasked inputs. Formally, the partial convolution at every location is:

$$x' = \begin{cases} \mathbf{W}^T(\mathbf{X} \odot \mathbf{M}) \frac{\mathbf{sum(1)}}{\mathbf{sum(M)}} + b, & \text{if } \mathbf{sum(M)} > 0 \\ 0, & \text{otherwise} \end{cases}$$

  where 1 has the same shape as M but with all elements being 1, the scaling factor sum(1)/sum(M) applies appropriate scaling to adjust for the varying amount of valid (unmasked) inputs.

  After each partial convolution operation, we then update our mask as follows: if the convolution was able to condition its output on at least one valid input value, then we mark that location to be valid. This is expressed as:

$$m' = \begin{cases} 1, & \text{if sum}(\mathbf{M}) > 0 \\ 0, & \text{otherwise} \end{cases}$$

With sufficient successive applications of the partial convolution layer, any mask will eventually be all ones, if the input contains any valid pixels.
given that the binary mask is only "1" and "0", we can update the mask in each iteration using conv2d such that the kernels weights are all "1" and without bias
(torch.nn.init.constant__(self.mask_conv.weight,1)).

- network architecture: They use UNet architecture, but replacing all convolutional layers with partial convolutional layers and using nearest neighbor upsampling in the decoding stage.
- Training:

  Input: Input image with hole I_{in}, initial binary mask M (0 for holes), the ground truth image I_{gt}

  Output: the network predictionI_{out}

- Losses:

1. **L1 Loss -** this loss is for ensuring the pixel wise reconstruction accuracy.

   L1 between the predicted output on the hole and the ground truth on the hole position, and L1 between the predicted output on the non-hole pixels and the ground truth on the non-hole pixels:

   $L_{hole} = \frac{1}{N_{I_{gt}}} || (1 - M) \odot (I_{out} - I_{gt})||_1 \quad L_{valid} = \frac{1}{N_{I_{gt}}} || M \odot (I_{out} - I_{gt})||_1$ where $N_{I_{gt}}$ denotes the number of elements in I_{gt}.

2. **Perceptual loss - we want the filled image and the ground truth image to have similar feature representations computed by a pre-trained network like VGG-16.**

   we compute perceptual loss between the a) the image output (I_{out}) and the image ground truth, b)the image output with the non-hole pixels directly set to the ground truth (I_{comp}) , and the image ground truth.
   Specifically, we feed the ground truth image and the filled image to a pre-trained VGG-16 to extract features. Then, we calculate the L1 distance between the feature values of I_{out} , I_{comp} and the ground truth at all or several layers (P layers in the equation). This perceptual is small when the completed image is semantically close to its ground truth image.

   $$\mathcal{L}_{perceptual} = \sum_{p=0}^{P-1} \frac{\|\Psi_p^{\mathbf{I}_{out}} - \Psi_p^{\mathbf{I}_{gt}}\|_1}{N_{\Psi_p^{\mathbf{I}_{gt}}}} + \sum_{p=0}^{P-1} \frac{\|\Psi_p^{\mathbf{I}_{comp}} - \Psi_p^{\mathbf{I}_{gt}}\|_1}{N_{\Psi_p^{\mathbf{I}_{gt}}}}$$

3. Style loss term

We further include the style-loss term, which is similar to the perceptual loss, but we first perform an autocorrelation (Gram matrix) on each feature map before applying the L1 . The Gram matrix contains the style information of an image such as textures and colors

$$\mathcal{L}_{style_{out}} = \sum_{p=0}^{P-1} \frac{1}{C_p C_p} \left|\left| K_p \left(\left(\Psi_p^{\mathbf{I}_{out}}\right)^{\intercal} \left(\Psi_p^{\mathbf{I}_{out}}\right) - \left(\Psi_p^{\mathbf{I}_{gt}}\right)^{\intercal} \left(\Psi_p^{\mathbf{I}_{gt}}\right) \right) \right|\right|_1$$

$$\mathcal{L}_{style_{comp}} = \sum_{p=0}^{P-1} \frac{1}{C_p C_p} \left|\left| K_p \left(\left(\Psi_p^{\mathbf{I}_{comp}}\right)^{\intercal} \left(\Psi_p^{\mathbf{I}_{comp}}\right) - \left(\Psi_p^{\mathbf{I}_{gt}}\right)^{\intercal} \left(\Psi_p^{\mathbf{I}_{gt}}\right) \right) \right|\right|_1$$

4. Total variation loss (TV) - smoothing penalty on R, where R is the region of 1-pixel dilation of the hole region. Simply speaking, this loss is adopted to ensure the smoothness of the completed images.

In total:

$$\mathcal{L}_{total} = \mathcal{L}_{valid} + 6\mathcal{L}_{hole} + 0.05\mathcal{L}_{perceptual} + 120(\mathcal{L}_{style_{out}} + \mathcal{L}_{style_{comp}}) + 0.1\mathcal{L}_{tv}$$

Free-Form Image Inpainting with Gated Convolution (2019)

Motivation: This work considers the problem of applying convolution to valid pixels and invalid pixels, and represents a problem with the solution of the previous approach, aka the partial convolution.

First, It heuristically classifies all spatial locations to be either valid or invalid. The mask in the next layer will be set to ones no matter how many pixels are covered by the filter range in the previous layer (for example, 1 valid pixel and 9 valid pixels are treated the same to update the current mask).

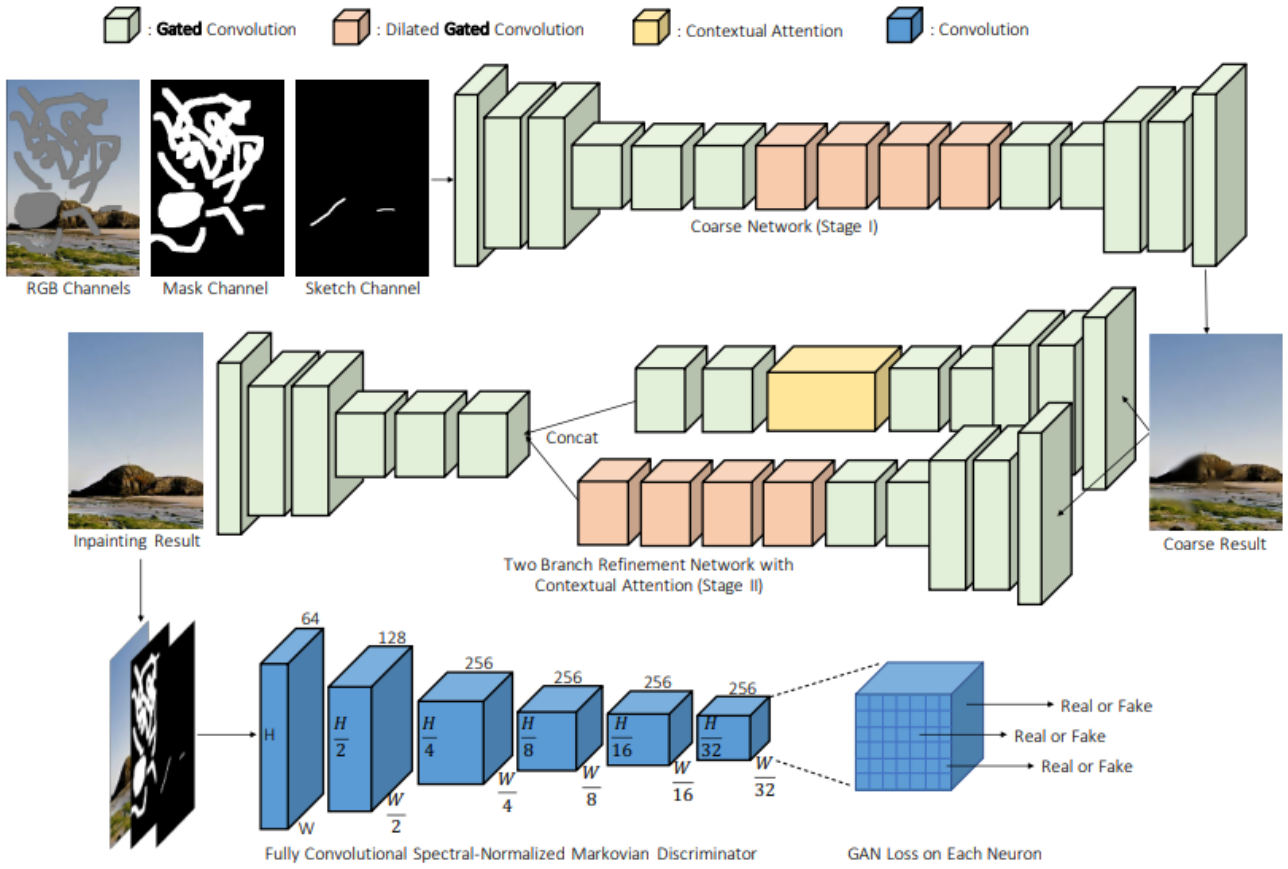
Second, if we extend to user-guided image inpainting where users provide sparse sketch inside the mask, should these pixel locations be considered as valid or invalid? How to properly update the mask for next layer?

Third, for partial convolution the “invalid” pixels will progressively disappear layer by layer and the rule-based mask will be all ones in deep layers. However, to synthesize pixels in the hole these deep layers may also need the information of whether current locations are inside or outside the hole. The partial convolution with all-ones mask cannot provide such information.

In this paper, they show that if we allow the network to learn the mask automatically (the mask that distinguishes between the valid pixels and the hole), the mask may have different values based on whether current locations are masked or not in the input image, even in deep layers.

Method

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



1. Gated Convolution -

   Instead of a hard-gating mask updated with rules (as in partial convolution), they propose gated convolution. Gated convolutions learn soft mask automatically from data. It is formulated as:

$$Gating_{y,x} = \sum\sum W_g \cdot I$$
$$Feature_{y,x} = \sum\sum W_f \cdot I$$
$$O_{y,x} = \phi(Feature_{y,x}) \odot \sigma(Gating_{y,x})$$

   where σis sigmoid function thus the output gating values are between zeros and ones. Φ can be any activation functions (for example, ReLU, ELU and LeakyReLU). W_g and W_f are two different convolutional filters.
   The proposed gated convolution learns a dynamic feature selection mechanism for each channel and each spatial location.

2. Spectral-Normalized Markovian Discriminator (SN-PatchGAN) -

For previous inpainting networks which try to fill a single rectangular hole, an additional local GAN is used on the masked rectangular region to improve results. However, we consider the task of free-form image inpainting where there may be multiple holes with any shape at any location.

The input to the Patch GAN discriminator is the concatenation of the image, the mask and the guidance channels. The output is a 3D feature map of shape CxHxW. Then, they directly apply GANs for each feature element in the feature map, formulating CxHxW number of GANs focusing on different locations (HxW patches) and different semantics (represented by the C channels). They also adapt spectral normalization to further stabilize the training of GANs.
Perceptual loss is not used since similar patch-level information is already encoded in SN-PatchGAN.
Compared with PartialConv in which 6 different loss terms and balancing hyper-parameters are used, our final objective function for inpainting network is only composed of pixel-wise L1 reconstruction loss and SN-PatchGAN loss, with default loss balancing hyper-parameter as 1 : 1.

3. The coarse network -
   simple encoder-decoder network. The input is the concatenation of the image with the mask (and also the sketch if given) and the output is an RGB coarse image of the same spatial size. All the convolution layers are replaced with gated convolutional layers.

```
 # img: entire img

 # mask: 1 for mask region; 0 for unmask region

 # 1 - mask: 1 for valid pixels, 0 for the mask region

 # img * (1 - mask): ground truth image with 0 on the unfilled
regions

 first_masked_img = img * (1 - mask) + mask

 first_in = torch.cat((first_masked_img, mask), 1)  # in: [B, 4, H,
W]

 first_out = self.coarse(first_in)      # out: [B, 3, H, W]
```

4. The refinement network -
   receives as input the image where the valid pixels are taken from the input image and the hole pixels are filled with the pixels from the coarse network, and also the mask.

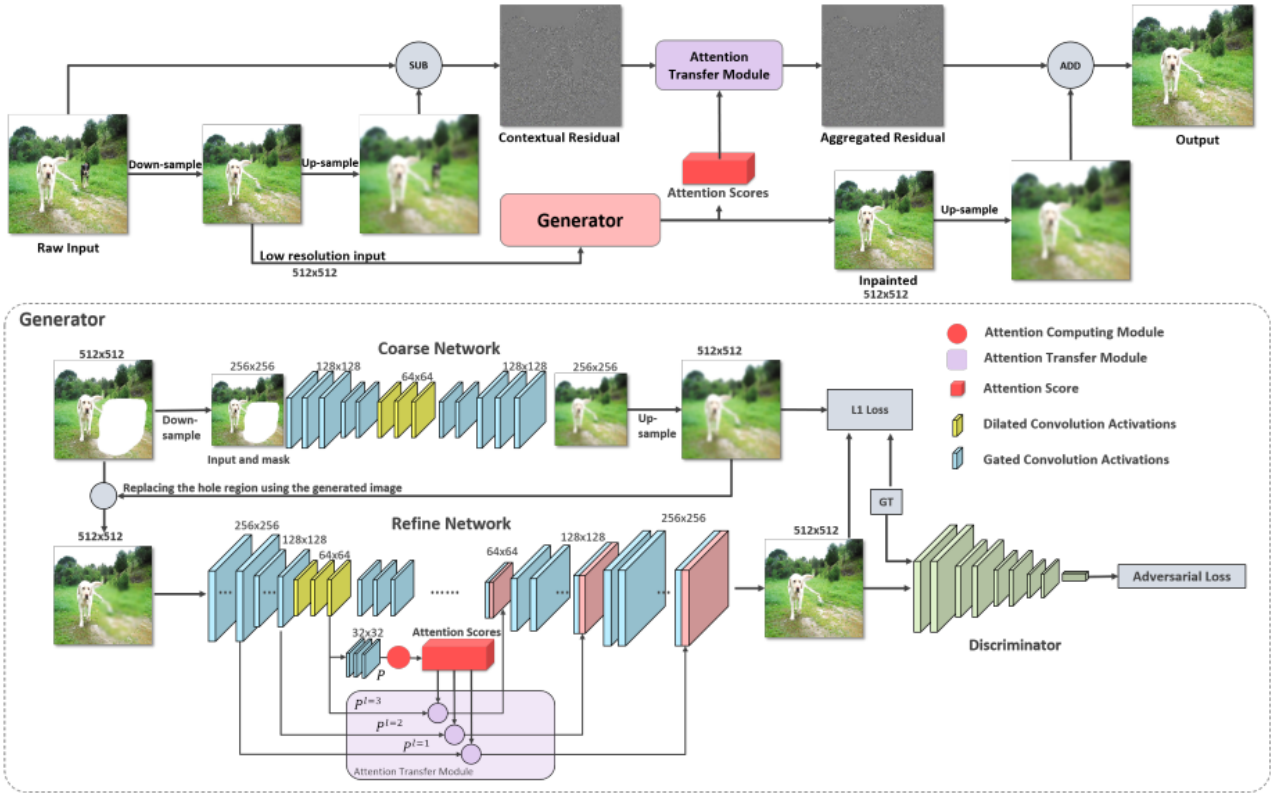
```
second_masked_img = img * (1 - mask) + first_out * mask

second_in = torch.cat((second_masked_img, mask), 1)  # in: [B, 4, H,
W]

second_out = self.refinement(second_in)    # out: [B, 3, H, W]
```

Mask generation

The algorithm to automatically generate free-form masks is important and non-trivial. The sampled masks, in essence, should be (1) similar to masks drawn in real use-cases, (2) diverse to avoid over-fitting, (3) efficient in

computation and storage, (4) controllable and flexible. We introduce a simple algorithm to automatically generate random free-form masks on-the-fly during training. For the task of hole filling, users behave like using an eraser to brush back and forth to mask out undesired regions. This behavior can be simply simulated with a randomized algorithm by drawing lines and rotating angles repeatedly. To ensure smoothness of two lines, we also draw a circle in joints between the two lines.

Contextual Residual Aggregation for Ultra High-Resolution Image Inpainting (2020)



Pipeline:

1) Given a high-resolution input image, we first down-sample the image to 512 × 512 and then up-sample it to obtain a blurry large image of the same size as the raw input.
2) The generator takes the low-resolution image and fills the holes.
3) Meanwhile, the attention scores are calculated by the Attention Computing Module (ACM) of the generator
4) Also, the contextual residuals are computed by subtracting the large blurry image from the raw input, and the aggregated residuals in the mask region are then calculated from the contextual residuals and attention scores through an Attention Transfer Module (ATM)

5) Finally, adding the aggregated residuals to the up-sampled inpainted result generates a large sharp output in the mask region while the area outside mask is simply a copy of the original raw input.

Training loss: 1) adversarial loss - WGAN-GP loss 2)L1 loss to force the consistency of the prediction with the original image (one for the in-hole pixels, and another for the valid pixels)

More works

- UCTGAN: Diverse Image Inpainting Based on Unsupervised Cross-Space Translation (2020)
- Recurrent Feature Reasoning for Image Inpainting (2020)
- Distillation-Guided Image Inpainting (2021)

Self Supervised Learning

Learning good image representations is a key challenge in computer vision as it allows for efficient training on downstream tasks. Many different training approaches have been proposed to learn such representations, usually relying on visual pretext tasks. Among them, state-of-the-art contrastive methods are trained by reducing the distance between representations of different augmented views of the same image (‘positive pairs’), and increasing the distance between representations of augmented views from different images (‘negative pairs’). These methods need careful treatment of negative pairs by either relying on large batch sizes, memory banks or customized mining strategies to retrieve the negative pairs. In addition, their performance critically depends on the choice of image augmentations.

Most unsupervised methods for representation learning can be categorized as either generative or discriminative.

Generative approaches to representation learning build a distribution over data and latent embedding and use the learned embeddings as image representations. Many of these approaches rely either on auto-encoding of images or on adversarial learning, jointly modeling data and representation. Generative methods typically operate directly in pixel space. This however is computationally expensive, and the high level of detail required for image generation may not be necessary for representation learning. Among discriminative methods, contrastive methods currently achieve state-of-the-art performance in self-supervised learning. Contrastive approaches avoid a costly generation step in pixel space by bringing the representation of different views of the same image closer (‘positive pairs’), and spreading representations of views from different images (‘negative pairs’) apart. Contrastive methods often require comparing each example with many other examples to work well prompting the question of whether using negative pairs is necessary

Deep clustering

DeepCluster - Advantage of using clustering - can be applied on any specific domain or dataset, like satellite or medical images, or on images captured with a new modality, like depth, where annotations are not always available in quantity. clustering has the advantage of requiring little domain knowledge and no specific signal from the inputs.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Alternating between clustering of the image descriptors and updating the weights of the convnet by predicting the cluster assignments. The idea of this work is to bootstrap the discriminative power of a convnet (power =their convolutional structure which gives a strong prior on the input signal. Even with random weights, the classification accuracy on imagenet is far above the chance level). We cluster the output of the convnet and use the subsequent cluster assignments as “pseudo-labels” to optimize Eq. (1). This deep clustering (DeepCluster) approach iteratively learns the features an

BYOL

Does not use negative pairs. They suspect that not relying on negative pairs is one of the leading reasons for its improved robustness. In particular, BYOL uses two neural networks, referred to as online and target networks, that interact and learn from each other. Starting from an augmented view of an image, BYOL trains its online network to predict the target network’s representation of another augmented view of the same image.

from a given representation, referred to as target, we can train a new, potentially enhanced representation, referred to as online, by predicting the target representation. From there, we can expect to build a sequence of representations of increasing quality by iterating this procedure, using subsequent online networks as new target networks for further training. In practice, BYOL generalizes this bootstrapping procedure by iteratively refining its representation, but using a slowly moving exponential average of the online network as the target network instead of fixed checkpoints.

BYOL uses two neural networks to learn: the online and target networks.

The online network is defined by a set of weights θ and is comprised of three stages: an encoder f_θ, a projector g_θ and a predictor q_θ.

The target network has the same architecture as the online network, but uses a different set of weights ξ. The target network provides the regression targets to train the online network, and its parameters ξ are an exponential moving average of the online parameters θ. More precisely, given a target decay rate $\tau \in [0, 1]$, after each training step we perform the following update, $\xi \leftarrow \tau \xi + (1 - \tau)\theta$.

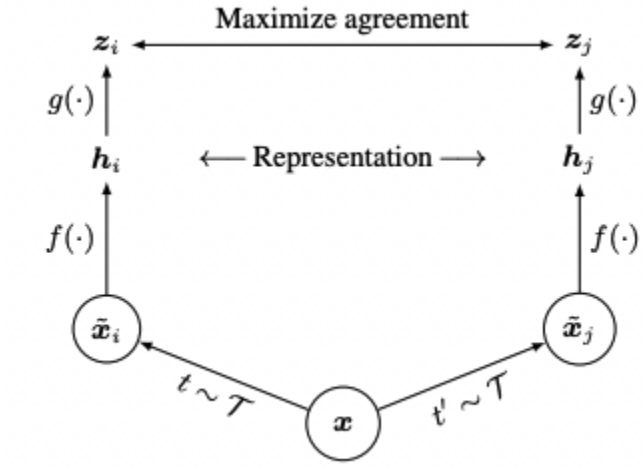
Given a set of images D, an image x ~ D sampled uniformly from D, and two distributions of image augmentations T and T’ , BYOL produces two augmented views v = t(x) and v’ = t’ (x) from x by applying respectively image augmentations t, t’.

SimCLR

SimCLR learns representations by maximizing agreement between differently augmented views of the same data example via a contrastive loss in the latent space. This framework comprise the following four major components:

1) A stochastic data augmentation module that transforms randomly an image into two correlated views of the image.
2) A neural network base encoder f(·) that extracts representation vectors from augmented data examples.

3) A small neural network projection head g(·) that maps representations to the space where contrastive loss is applied.
4) A contrastive loss function defined for a contrastive prediction task. Given a set $\{x_k\}$ including a positive pair of examples x_i and x_j, the contrastive prediction task aims to identify x_j in $\{x_k\}_{k \neq i}$ for a given x_i.



Vision Transformers

Motivation

Recently, Transformers have been viewed as a strong alternative to Convolutional Neural Networks (CNNs) in visual recognition tasks such as classification and detection. Unlike the convolution operation in CNNs, which has a limited receptive field, the self-attention mechanism in the Transformers can capture the long-distance information and dynamically adapt the receptive field according to the image content. As a result, the Transformers are considered more flexible and powerful than CNNs, thus being promising to achieve more progress in visual recognition.

Disadvantages of Transformers for image data

- Unlike word tokens in NLP, visual elements have various scales. Especially in object detection.
- Images are in much higher resolution and vision tasks that require pixel-level predictions are intractable for transformers because the computation complexity(+memory usage) of self-attention is quadratic to image size.
- ViT lacks certain desirable properties inherently built into the CNN architecture that make CNNs uniquely suited to solve vision tasks. images have a strong 2D local structure: spatially neighboring pixels are usually highly correlated. The CNN architecture forces the capture of this local structure by using local receptive fields, shared weights, and spatial subsampling, and thus also achieves some degree of shift, scale, and distortion invariance. In addition, the hierarchical structure of convolutional kernels learns visual patterns

that take into account local spatial context at varying levels of complexity, from simple low-level edges and textures to higher-order semantic patterns.

- ViT does not benefit from the built-in locality bias that is present in CNNs: neighbouring pixels within a patch may be highly correlated, but this bias is not encoded into the ViT architecture. That is, ViT encodes inter-patch correlations well, but not intra-patch correlations. Further, each image may require a different patch size and location, depending on the size and location of objects in the image.

Advantages of Vision transformers

- dynamic attention, global context fusion, and better generalization

Operations in CNN that should be integrated into vision transformers

- Spatial downsampling while concurrently increasing the number of feature maps
- The features of ResNet capture the desired local structure (edges, lines, textures, etc.) progressively from the bottom layer (conv1) to the middle layer (conv25).
  CNN architectures begin by applying convolutional layers of a small receptive field for low-level features, resulting in local dependencies between neighboring image regions. As processing continues and features become more semantic, the effective receptive field is gradually increased, capturing longer-ranged dependencies.

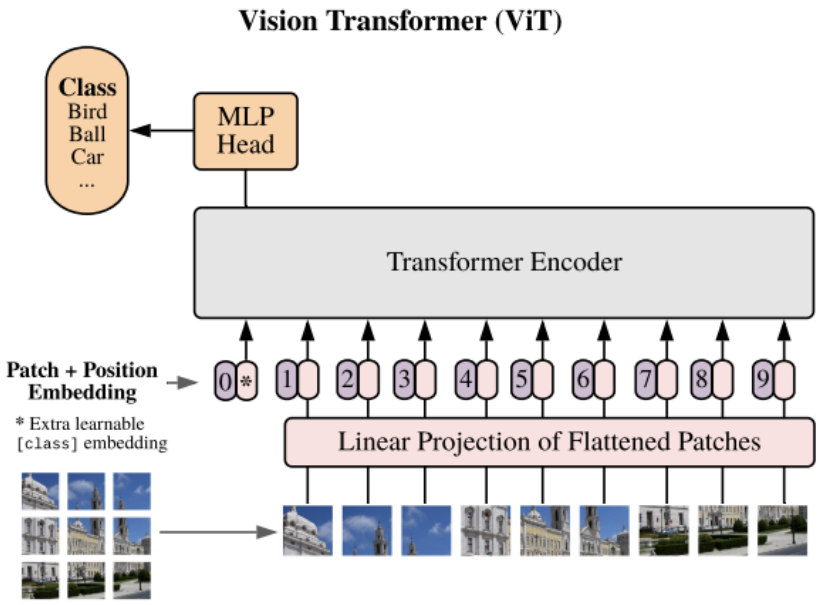
ViT: Vision Transformer

The paper's main goal was to show that a vanilla Transformer, once adapted to deal with data from the visual domain, could compete with some of the most performant convolutional neural networks (CNNs) developed up to that point.
The Vision Transformer architecture is conceptually simple: divide the image into patches (each patch of size 16x16 → total 14x14 patches), flatten and project them (using linear layer) into a D-dimensional embedding space obtaining the so-called patch embeddings. The transformer block described above is invariant to the order of the patch embeddings and thus does not consider their relative position. Thus, adding positional embeddings (a set of learnable vectors allowing the model to retain positional information) and concatenating a (learnable) class token, then let the Transformer encoder do its magic. Finally, a classification head is applied to the class token to obtain the model's logits.
The model's performance was acceptable when trained on ImageNet (1M images), great when pre-trained on ImageNet-21k (14M images), and state-of-the-art when pre-trained on Google's internal JFT-300M dataset (300M images).

The striking performance improvement was due to the reduced inductive bias that characterizes Vision Transformers. By making fewer assumptions about the data, Vision Transformers could better adapt themselves to the given task. However, this ability came at a cost – when the sample size was too small (such as in the ImageNet case), the models overfit, resulting in degraded performance.

The goal for many follow-up papers would be that of matching (and surpassing) the performance of the best convolutional models in the "small" data regime – ImageNet (which is after all over a million images) and below.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



Architecture

1) **Multi-head self-attention layers (MSA).**

The attention mechanism is based on a trainable associative memory with (key, value) vector pairs. A query vector q $\in R^d$ is matched against a set of k key vectors (packed together into a matrix K $\in R^{k \times d}$) using inner products. These inner products are then scaled and normalized with a softmax function to obtain k weights. The output of the attention is the weighted sum of a set of k value vectors (packed into V $\in R^{k \times d}$). For a sequence of N query vectors (packed into Q $\in R^{N \times d}$), it produces an output matrix (of size N × d):

$$\text{Attention}(Q, K, V) = \text{Softmax}(QK^\top/\sqrt{d})V,$$

where the Softmax function is applied over each row of the input matrix and the $\sqrt{}$ d term provides appropriate normalization.

Query, key and values matrices are themselves computed from a sequence of N input vectors (packed into X $\in R^{N \times d}$):

Q = XWQ, K = XWK, V = XWV, using linear transformations WQ, WK, WV with the constraint k = N, meaning that the attention is in between all the input vectors.

Finally, Multi-head self-attention layer (MSA) is defined by considering h attention "heads", ie h self-attention functions applied to the input. Each head provides a sequence of size N × d. These h sequences are rearranged into a N × dh sequence that is reprojected by a linear layer into N × D.

2) **Transformer Block**
To get a full transformer block, we add a Feed-Forward Network (FFN) on top of the MSA layer. This FFN is composed of two linear layers separated by a GeLu activation. The first linear layer expands the dimension from D to 4D, and the second layer reduces the dimension from 4D back to D. Both MSA and FFN are operating as residual operators thank to skip-connections, and with a layer normalization.

3) **The class token**
a learnable embedding, appended to the sequence of embedded patches before the first layer goes through the transformer layers and is then projected with a linear layer (MLP) to predict the class. The final embedding of the CLS token (after all the transformer blocks and before the MLP) serves as the image representation. The transformer thus process batches of (N + 1) tokens of dimension D, of which only the class vector is used to predict the output. This architecture forces the self-attention to spread information between the patch tokens and the class token.

Complexity

$$\Omega(\mathrm{MSA}) = 4hwC^2 + 2(hw)^2C,$$

Observations

1) The granularity of the patch size affects the accuracy and complexity of ViT; with fine-grained patch size, ViT can perform better but results in higher FLOPs and memory consumption. For example, the ViT with a patch size of 16 outperforms the ViT with a patch size of 32 by 6% but the former needs 4× more FLOPs

DeiT: Training data-efficient image transformers & distillation through attention

The first paper to show that models based on ViTs could be competitive on ImageNet without access to additional data. The main architecture of DeiT is very similar to the architecture of ViT (the transformer blocks, the class token, the positional embedding and the patch embedding is the same).

Two main contributions of the paper:

1) Distillation through attention

In this approach, they assume access to a strong image classifier as a teacher model. Interestingly, the authors found CNNs to be better teacher networks than other Vision Transformers. The fact that the convnet is a better teacher is probably due to the inductive bias inherited by the transformers through distillation, as explained in Abnar et al. (Transferring inductive biases through knowledge distillation).Here, an additional learnable token, called the distillation token is concatenated to the patch embeddings and the class embedding. The distillation token is used similarly as the class token: it interacts with other embeddings through self-attention, and is output by the network after the last layer. Its target objective is given by the distillation component of the loss.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

- Soft distillation (not used)

  Minimizes the KL divergence between the softmax of the teacher and the softmax of the student model. Let Z_t be the logits of the teacher model, Z_S the logits of the student model. We denote by τ the temperature for the distillation, λ the coefficient balancing the Kullback–Leibler divergence loss (KL) and the cross-entropy (LCE) on ground truth labels y, and ψ the softmax function. The distillation objective is:

$$\mathcal{L}_{\text{global}} = (1-\lambda)\mathcal{L}_{\text{CE}}(\psi(Z_{\text{s}}), y) + \lambda\tau^2 \text{KL}(\psi(Z_{\text{s}}/\tau), \psi(Z_{\text{t}}/\tau)).$$

- Hard label distillation

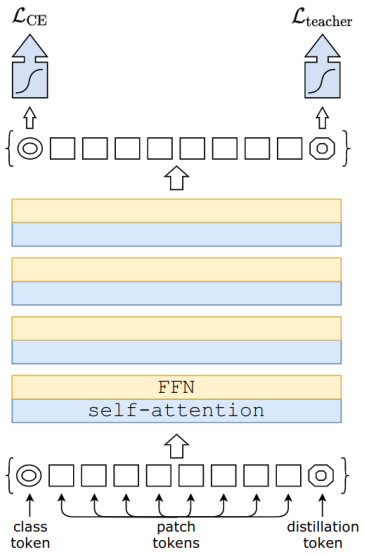
  We introduce a variant of distillation where we take the hard decision of the teacher as a true label. Let $y_{teacher} = \ argmax_c Z_t(c)$ be the hard decision of the teacher, the objective associated with this hard-label distillation is:

$$\mathcal{L}_{hardDistill} = \frac{1}{2}\mathcal{L}_{CE}(\sigma(Z_{cls}), y_{true}) + \frac{1}{2}\mathcal{L}_{CE}(\sigma(Z_{distill}), y_{teacher})$$

  where σ is the softmax function, Z_{CLS} and $Z_{distill}$ are the student model's logits derived respectively from the class and distillation tokens, and y_{true} and $y_{teacher}$ are respectively the true and the teacher's hard labels. This distillation technique allows the model to learn even when the combination of multiple strong data augmentations causes the provided label to be imprecise, as the teacher network will produce the most probable label.

  For a given image, the hard label associated with the teacher may change depending on the specific data augmentation. We will see that this choice is better than the traditional one, while being parameter-free and conceptually simpler: The teacher prediction y_t plays the same role as the true label y. Note also that the hard labels can also be converted into soft labels with label smoothing, where the true label is considered to have a probability of 1 − ε, and the remaining ε is shared across the remaining classes. We fix this parameter to ε = 0.1 in our all experiments that use true labels.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



2) A novel training recipe

- Initialization - truncated normal distribution.
- Data augmentation - transformers require a larger amount of data. Thus, in order to train with datasets of the same size, we rely on extensive data augmentation. The augmentations that are used are: Rand-Augment, random erasing.
- Regularization - stochastic depth (which facilitates the convergence of transformers), Mixup, CutMix, repeated augmentations.
- Exponential Moving Average (EMA)

CPVT: Conditional Positional Encodings for Vision Transformers

This paper studied alternatives to the positional embeddings and class token used in ViTs. In particular, the paper proposes the use of Positional Encodings Generators (PEGs), a module that produces positional encodings dynamically, and the use of global average pooling as an alternative to the (non-translation-invariant) class token.

- Translation invariance - For example, in the classification task, the location of the target in an image should not alter the network's activations so as to have the same classification result.

The self-attention operation in Transformers is permutation-invariant, which cannot leverage the order of the tokens in an input sequence.

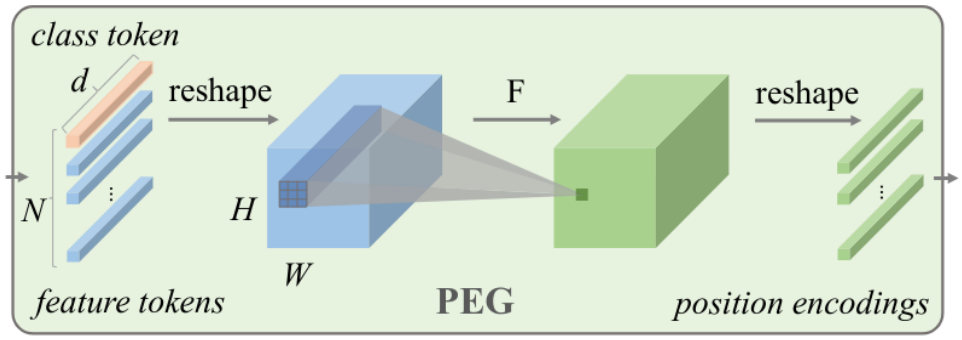
The problem of positional embedding used in DeiT and ViT:

Previous works add the absolute positional encoding to each token in the input sequence - the positional encoding can either be learnable or fixed with sinusoidal functions of different frequencies. Despite being effective and noninvasive, these positional encodings seriously harm the flexibility of the Transformers, hampering their broader applications. Taking the learnable version as an example, the encodings are often a vector of equal length to the input sequence, which are jointly updated with the network weights during training.

Thus, the length and the value of the positional encodings are fixed once trained. During testing, it causes difficulty handling the sequences longer than the ones in the training data.

Conditional Position encoding - key ideas

The proposed PE is conditioned on the local neighborhood of input tokens and is adaptable to arbitrary input sizes, which enables processing images of much larger resolutions. Characterizing the local relationship by positional encodings is permutation-variant because the permutation of input sequences also affects the order in some local neighborhoods. In contrast, the translation of an object in an input image might not change the order in its local neighborhood, i.e., translation-invariant.



To condition on the local neighbors, we first reshape the flattened input sequence X $\in R^{B\times N\times C}$ of DeiT back to $X' \in R^{B\times H\times W\times C}$ in the 2-D image space. Then, a function (denoted by F in Figure) is repeatedly applied to the local patch in X' to produce the conditional positional encodings $E^{B\times H\times W\times C}$. PEG can be efficiently implemented with a 2-D convolution with kernel k (k ≥ 3) and (k−1)/2 zero paddings. Note that the zero paddings here are important to make the model be aware of the absolute positions, and F can be of various forms such as separable convolutions and many others.

TNT: Transformer in Transformer

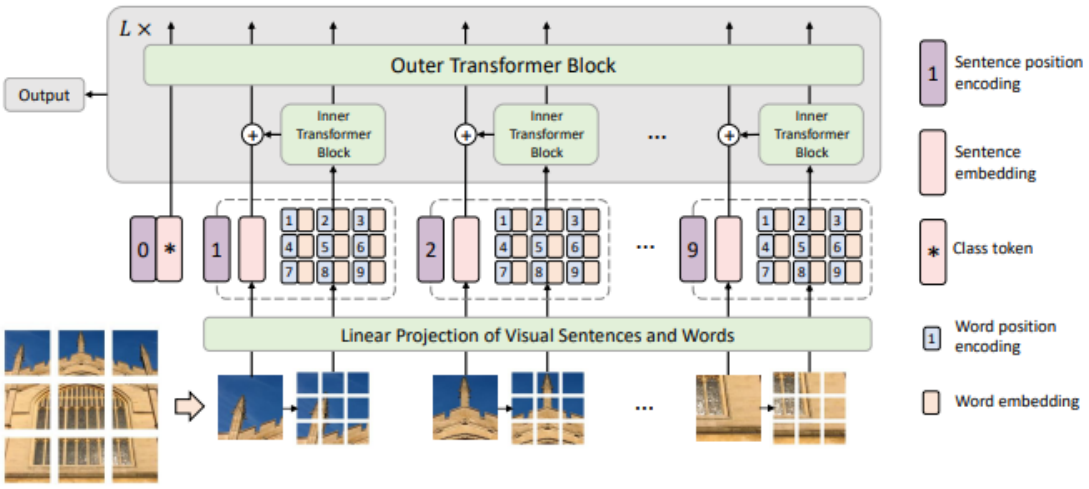
Goal: learn both global and local information in an image.

Basically, the visual transformers first divide the input images into several local patches and then calculate both representations and their relationship. Since natural images are of high complexity with abundant detail and color information, the granularity of the patch dividing is not fine enough for excavating features of objects in different scales and locations.

Key idea of this work -

In this paper, we point out that the attention inside these local patches are also essential for building visual transformers with high performance and we explore a new architecture, namely, Transformer iN Transformer (TNT). Specifically, we regard the local patches (e.g., 16×16) as "visual sentences" and present to further divide them into smaller patches (e.g., 4×4) as "visual words". The attention of each word will be calculated with other words in the given visual sentence with negligible computational costs. Features of both words and sentences will be aggregated to enhance the representation ability.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



1. Divide the images to patches as ViT - “visual sentences”
2. Divide each visual sentence into sub-patched - “visual words”
3. Create independently sentence embeddings and word embeddings
4. Each T2T block is composed from 2 processes and networks. In each TNT block:
    a. For the word embeddings, we utilize a transformer block to explore the relation between visual words. This is inner transformer block. This process builds the relationships among visual words by computing interactions between any two visual words.
    b. The updated word embeddings projected to the domain of sentence embedding and added to the sentence embedding
    c. Use the standard transformer block for transforming the sentence embeddings (attention on the updated sentences)

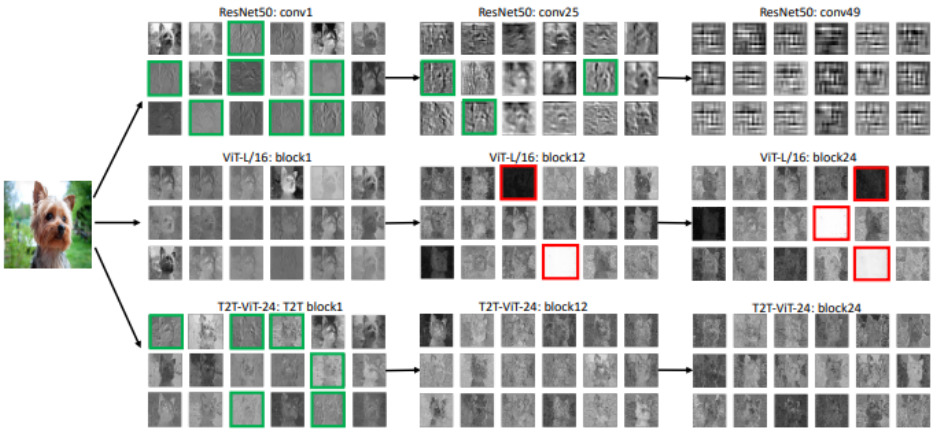
T2T ViT: Training Vision Transformers from Scratch on ImageNet

Though ViT proves the full-transformer architecture is promising for vision tasks, its performance is still inferior to that of similar-sized CNN counterparts (e.g. ResNets) when trained from scratch on a midsize dataset (e.g., ImageNet). We hypothesize that such performance gap roots in two main limitations of ViT:

1) the straightforward tokenization of input images by hard split makes ViT unable to model the image local structure like edges and lines, and thus it requires significantly more training samples (like JFT-300M for pretraining) than CNNs for achieving similar performance;

2) the attention backbone of ViT is not well designed as CNNs for vision tasks, which contains redundancy and leads to limited feature richness and difficulties in model training. To verify our hypotheses, we conduct a pilot study to investigate the difference in the learned features of ViTL/16 [12] and ResNet50 [15] through visualization in Fig. 2. We observe the features of ResNet capture the desired local structure (edges, lines, textures, etc.) progressively from the bottom layer (conv1) to the middle layer (conv25). However, the features of ViT are quite different: the structure information is poorly modeled while the global relations (e.g., the whole dog) are captured by all the attention blocks. These observations indicate that the vanilla ViT ignores the local structure

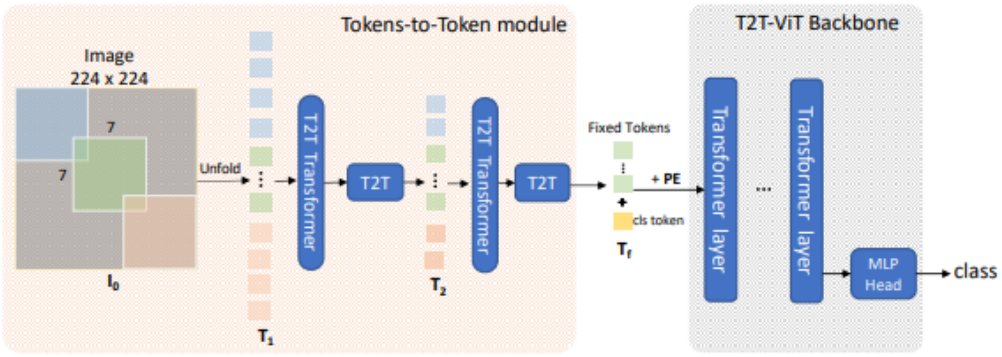
when directly splitting images to tokens with fixed length. Besides, we find many channels in ViT have zero value (highlighted in red in Fig. 2), implying the backbone of ViT is not efficient as ResNets and offers limited feature richness when training samples are not enough.



The Token-to-Token (T2T) module aims to overcome the limitation of simple tokenization in ViT. It progressively structurizes an image to tokens and models the local structure information, and in this way the number of tokens can be reduced iteratively. Each T2T process has two steps: Restructurization and Soft Split (SS):

1. Re-structurization - given a sequence of tokens T from the preceding transformer layer, it will be transformed by the self-attention block (regular transformer block). Then the transformers tokens will be reshaped as an image in the spatial dimension.
2. Soft split - split the re-structurized image into patches with overlapping (just unfolding). As such, each patch is correlated with surrounding patches to establish a prior that there should be stronger correlations between surrounding tokens.

After 3 iterations of unfolding -> attention they take the resulted tokens and pass it to a ViT.



Disclaimer: Tokens-to-Token (T2T) mainly improves tokenization in ViT by concatenating multiple tokens within a sliding window into one token. However, this operation fundamentally differs from convolutions especially in

normalization details, and the concatenation of multiple tokens greatly increases complexity in computation and memory

PVT: Pyramid Vision Transformer: A Versatile Backbone for Dense Prediction without Convolutions

Key: global receptive field, using pyramid with smaller patch size for dense prediction tasks (as object detection)

Unlike the recently proposed Vision Transformer (ViT) that was designed for image classification specifically, we introduce the Pyramid Vision Transformer (PVT), which overcomes the difficulties of porting Transformer to various dense prediction tasks (=tasks that produce pixel level predictions as semantic segmentation, object detection and instance segmentation). Although ViT is applicable to image classification, it is challenging to directly adapt it to pixel-level dense predictions such as object detection and segmentation, because (1) its output feature map is single-scale and low-resolution (ignoring pixels, using attention on 32x32 patches), and (2) its computational and memory costs are relatively high even for common input image sizes.

PVT overcomes the difficulties of the conventional Transformer by (1) taking fine-grained image patches (i.e., 4×4 pixels per patch) as input to learn high-resolution representation, which is essential for dense prediction tasks; (2) introducing a progressive shrinking pyramid to reduce the number of tokens of Transformer as the network deepens, significantly reducing the computational cost, and (3) adopting a spatial-reduction attention (SRA) layer to further reduce the resource consumption when learning high-resolution features.

Advantages: (1) compared to the traditional CNN backbones, which have local receptive fields that increase with the network depth, our PVT always produces a global receptive field, which is more suitable for detection and segmentation. (2) compared to ViT, thanks to its advanced pyramid structure, our method can more easily be plugged into many representative dense prediction pipelines, e.g., RetinaNet [39] and Mask R-CNN [21]. Thirdly, we can build a convolution free pipeline by combining our PVT with other task-specific Transformer decoders, such as PVT+DETR [6] for object detection. To our knowledge, this is the first entirely convolution-free object detection pipeline.

Pipeline:

1) Given an input image, divide it to patches, each of size 4x4x3. Then, feed the flattened patches to a linear projection. And obtain embedded patches.
2) The embedded patches (with the positional embedding) are passed through a transformer block, and then the output is reshaped to a feature map (reshape the output in the spatial dimension).
3) We divide the obtained feature map from 2) to patches as in stage 1) and then passing it through the transformer block. Wo do it in total 4 times (4 stages in the pyramid).

Since the patch size is smaller than the patch size in Vit, the number of patches is huge! Thus, they change the Multi head attention procedure - they introduce spatial-reduction attention (SRA) for reducing the computational/memory overhead. They decreased the number of patches in K,V using conv2d by factor of R. The computational cost of the new attention operation is R times lower then those of MHA.

Simple example for shapes:

Reduction = 2
Q -> (N,C)
K, V -> (N/2, C)
Attn -> (N,C) * (C, N/2) = (N, N/2)
Output -> (N,N/2) * (N/2, C) = (N, C)

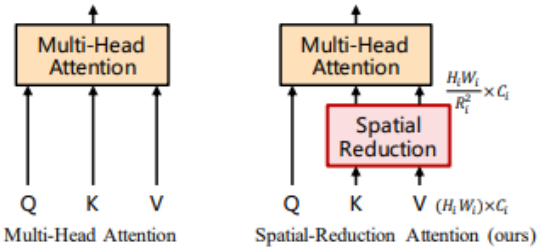


Figure 4: **Multi-head attention (MHA) *vs.* spatial-reduction attention (SRA).** With the spatial-reduction operation, the computational/memory cost of our SRA is much lower than that of MHA.

Swin Transformer: Hierarchical Vision Transformer using Shifted Windows

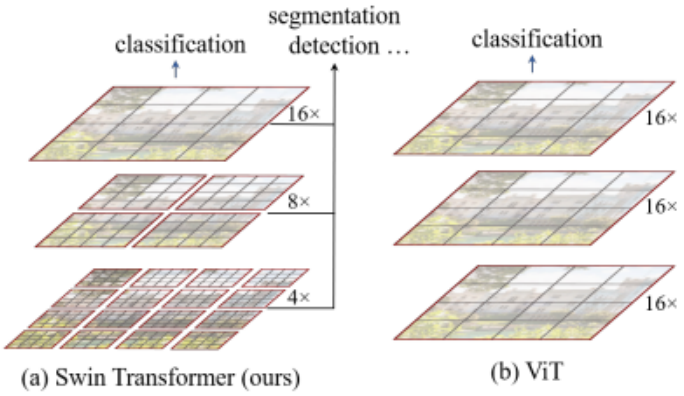
Motivation: In existing Transformer-based models, tokens are all of a fixed scale, a property unsuitable for several vision applications such as object detection. Another difference is the much higher resolution of pixels in images compared to words in passages of text. There exist many vision tasks such as semantic segmentation that require dense prediction at the pixel level, and this would be intractable for Transformer on high-resolution images, as the computational complexity of its self-attention is quadratic to image size.
The standard Transformer architecture [64] and its adaptation for image classification [20] both conduct global self attention, where the relationships between a token and all other tokens are computed. The global computation leads to quadratic complexity with respect to the number of tokens, making it unsuitable for many vision problems requiring an immense set of tokens for dense prediction or to represent a high-resolution image.

Swin transformer tries to overcome these problems by (1) Hierarchical representation by starting from small-sized patches and gradually increasing the size through merging to achieve scale-invariance (2) taking fine-grained image patches (i.e., 4×4 pixels per patch) as input to learn high-resolution representation, which is essential for dense prediction tasks;

Complexity: linear computational complexity to image size.

Swin Transformer constructs a hierarchical representation by starting from small-sized patches and gradually merging neighboring patches in deeper Transformer layers.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



The linear computational complexity is achieved by computing self-attention locally within non-overlapping windows that partition an image. The number of patches in each window is fixed, and thus the complexity becomes linear to image size.

A key design element of Swin Transformer is its shift of the window partition between consecutive self-attention layers. The shifted windows bridge the windows of the preceding layer, providing connections among them that significantly enhance modeling power.

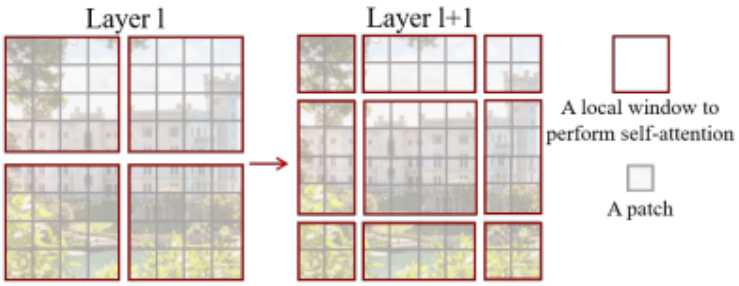


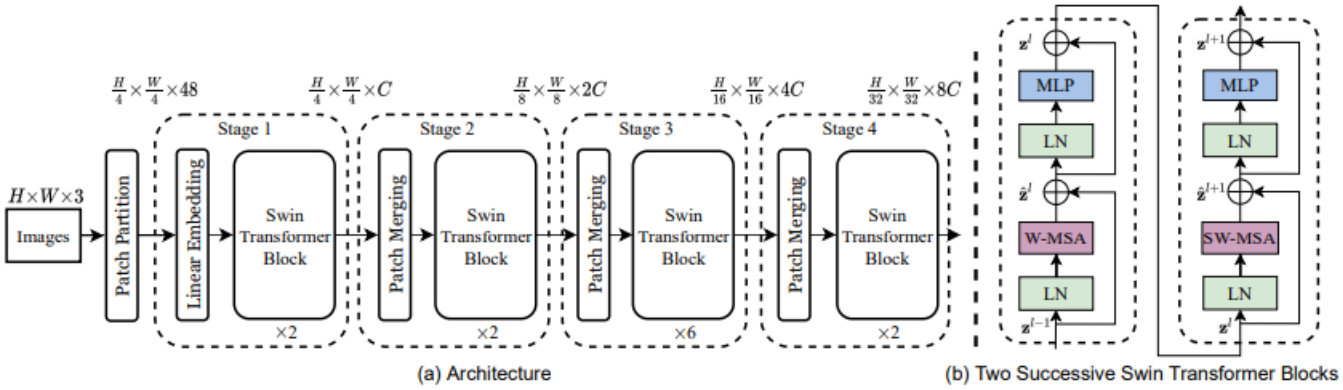
Figure 2. An illustration of the *shifted window* approach for computing self-attention in the proposed Swin Transformer architecture. In layer l (left), a regular window partitioning scheme is adopted, and self-attention is computed within each window. In the next layer $l + 1$ (right), the window partitioning is shifted, resulting in new windows. The self-attention computation in the new windows crosses the boundaries of the previous windows in layer l, providing connections among them.

Pipeline:

1) Split an RGB image into non-overlapping patches (each patch of size 4x4) and apply linear embedding layer to dimension C.
2) Several Transformer blocks with modified self-attention computation (Swin Transformer blocks) are applied on these patch tokens. The Transformer blocks maintain the number of tokens (H\4 × W\4), and together with the linear embedding are referred to as “Stage 1”.

3) To produce a hierarchical representation, the number of tokens is reduced by patch merging layers as the network gets deeper. The first patch merging layer concatenates the features of each group of 2 × 2 neighboring patches, (resulted in H\8 × W\8 tokens, each token with dimension 4C) and applies a linear layer on the 4C-dimensional concatenated features. This reduces the number of tokens by a multiple of 2×2 = 4 (2× downsampling of resolution), and the output dimension is set to 2C.
4) Swin Transformer blocks are applied afterwards for feature transformation, with the resolution kept at H\8 × W\8 . This first block of patch merging and feature transformation is denoted as “Stage 2”.
5) The procedure is repeated twice, as “Stage 3” and “Stage 4”, with output resolutions of H\16 × W\16 and H\32 × W\32 , respectively.

In the context of CNNs, we can intuitively understand the merging layers as pooling and transformer blocks as convolution layers. This approach enables the network to detect objects of various scales efficiently.



(a) Architecture (b) Two Successive Swin Transformer Blocks

Swin transformer block:

The problem with small patch size is the high computational complexity. To solve this problem, they propose to compute self attention within local windows. The windows are arranged to evenly partition the image in a non-overlapping manner. Supposing each window contains M × M patches, the computational complexity of a global MSA module and a window based one on an image of h × w patches are:

$$\Omega(\mathbf{MSA}) = 4hwC^2 + 2(hw)^2C,$$
$$\Omega(\mathbf{W\text{-}MSA}) = 4hwC^2 + 2M^2hwC,$$

The window-based self-attention module lacks connections across windows, which limits its modeling power. To introduce cross-window connections while maintaining the efficient computation of non-overlapping windows, we propose a shifted window partitioning approach which alternates between two partitioning configurations in consecutive Swin Transformer blocks.

The first module uses a regular partitioning strategy - partition the image to windows regularly and apply self attention within each window. Next - shift the windows by (M/2, M/2) pixels and compute self attention within the shifted windows.

Positional encoding:

They include a relative position bias $B \in R^{M^2 \times M^2}$ to each head in computing self attention where M^2 is number of patches in a window.

$$\text{Attention}(Q, K, V) = \text{SoftMax}(QK^T/\sqrt{d} + B)V,$$

Since the relative position along each axis lies in the range [−M + 1, M −1], we parameterize a smaller-sized bias matrix Bˆ $\in R^{(2M-1)\times(2M-1)}$, and values in B are taken from Bˆ.

Disadvantages: Although it can significantly reduce the complexity, it lacks the connections between different windows and thus results in a limited receptive field.

CrossViT: Cross-Attention Multi-Scale Vision Transformer for Image Classification

Motivation: learn multi-scale feature representations in transformer models

for image recognition. Their proposed approach is trying to leverage the advantages from more fine-grained patch sizes while balancing the complexity

Key idea: propose a dual-branch transformer to combine image patches (i.e., tokens in a transformer) of different sizes to produce stronger image features. Their approach processes small-patch and large-patch tokens with two separate branches of different computational complexity and these tokens are then fused purely by attention multiple times to complement each other. Furthermore, to reduce computation, they develop a simple yet effective token fusion module based on cross attention, which uses a single token

for each branch as a query to exchange information with other branches. The proposed cross-attention only requires linear time for both computational and memory complexity instead of quadratic time otherwise.

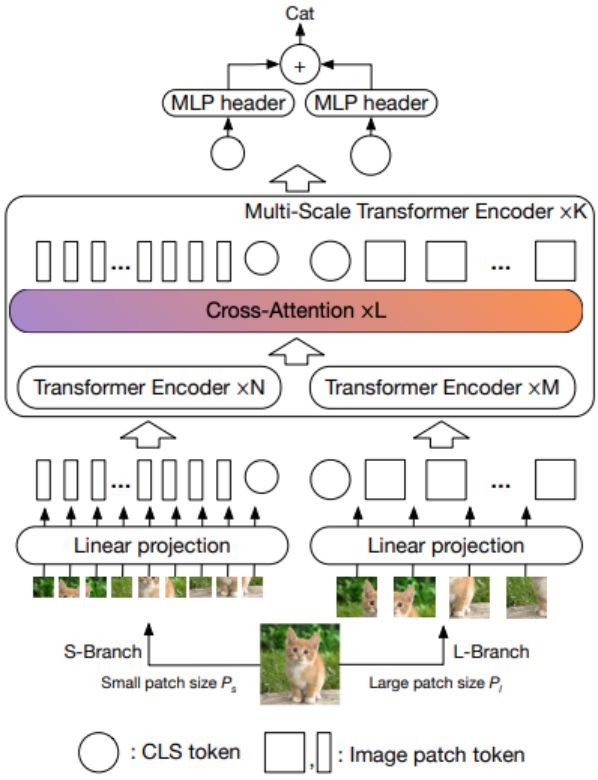
Method:

Our model is primarily composed of K multiscale transformer encoders where each encoder consists of two branches:

1) L-Branch: a large (primary) branch that utilizes coarse-grained patch size (P_l) with more transformer encoders and wider embedding dimensions
2) S-Branch: a small (complimentary) branch that operates at fine-grained patch size (Ps) with fewer encoders and smaller embedding dimensions

After processing the image tokens of different sizes in the L-Branch and the S-branch, they fuse the tokens at the end by an efficient module based on cross attention of the CLS tokens.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

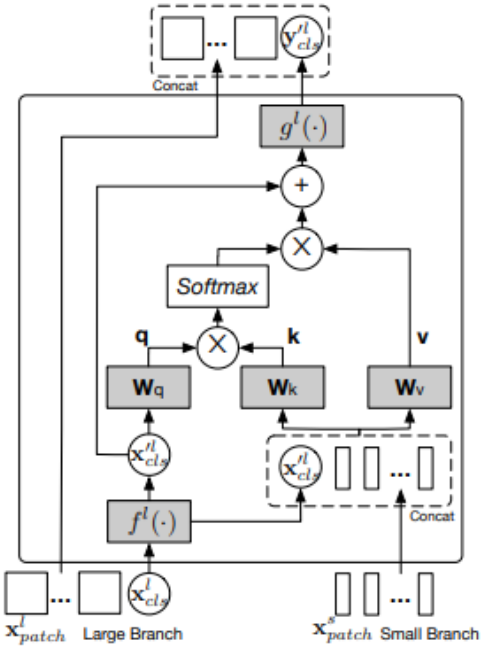


Cross attention Fusion - the fusion involves the CLS token of one branch and patch tokens of the other branch. Since the CLS token already learns abstract information among all patch tokens in its own branch, interacting with the patch tokens at the other branch helps to include information at a different scale. After the fusion with other branch tokens, the CLS token interacts with its own patch tokens again at the next transformer encoder, where it is able to pass the learned information from the other branch to its own patch tokens, to enrich the representation of each patch token. As the only use the CLS token as query, the cross attention is linear in the image size.

The pipeline of cross attention fusion (for the large branch. The same procedure is applied to the small branch)

1) Concat the patch tokens from the S-branch to the CLS token of the L-branch.
2) Apply cross attention between the CLS token of the L-branch (Q) and the concatenated tokens from 1) (K,V).
3) Add the result from 2) to the CLS token of the L-branch to produce a new CLS token.
4) The output: concat the new CLS token to the patch embedding of L-branch and repeat the whole process again.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



CvT: Introducing Convolutions to Vision Transformers

In this work, they study how to combine CNNs and Transformers to model both local and global dependencies for image classification in an efficient way.

Motivation: The authors hypothesize that convolutions can be strategically introduced to the ViT structure to improve performance and robustness, while concurrently maintaining a high degree of computational and memory efficiency. The CvT design introduces convolutions to two core sections of the ViT architecture:

1) convolutional token embedding - before each stage in the hierarchical transformer, they perform an overlapping convolution operation with stride on a 2-D reshaped token map (i.e reshaping flattened token sequences back to the spatial grid). This allows the model to not only capture local information, but also progressively decrease the sequence length while simultaneously increasing the dimension of token features across stages, achieving spatial downsampling while concurrently increasing the number of feature maps, as is performed in CNNs .
   This convolution operation in CvT aims to model local spatial contexts, from low-level edges to higher order semantic primitives, over a multi-stage hierarchy approach, similar to CNNs.
2) Convolutional projection - the linear projection prior to every self-attention block in the Transformer module is replaced with our proposed convolutional projection, which employs a s × s depth-wise separable convolution operation on an 2D-reshaped token map. This allows the model to further capture local spatial context and reduce semantic ambiguity in the attention mechanism. It also permits management of computational complexity, as the stride of convolution can be used to subsample the key and value matrices to improve efficiency by 4× or more, with minimal degradation of performance.

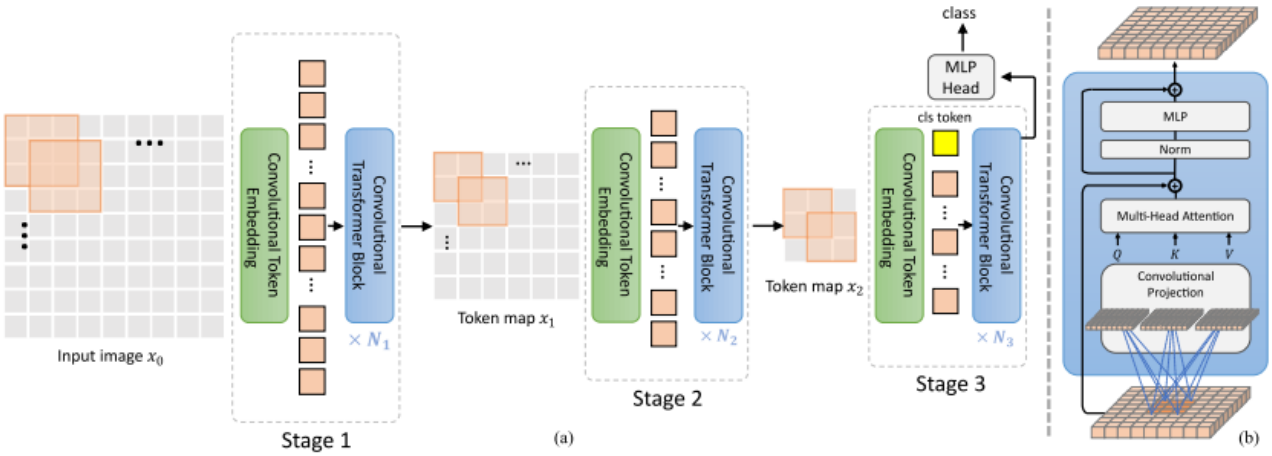
Method

multi-stage hierarchy design borrowed from CNNs is employed, where three stages in total are used in this work. Each stage has two parts:

1) the input image (or 2D reshaped token maps) are subjected to the Convolutional Token Embedding layer, which is implemented as a convolution with overlapping patches with tokens reshaped to the 2D spatial grid as the input (the degree of overlap can be controlled via the stride length). An additional layer normalization is applied to the tokens. This allows each stage to progressively reduce the number of tokens (i.e. feature resolution) while simultaneously increasing the width of the tokens (i.e. feature dimension), thus achieving spatial downsampling and increased richness of representation, similar to the design of CNNs.

2) a stack of the proposed Convolutional Transformer Blocks comprise the remainder of each stage. A depth-wise separable convolution operation referred to as Convolutional Projection, is applied for query, key, and value embeddings respectively, instead of the standard position-wise linear projection in ViT.

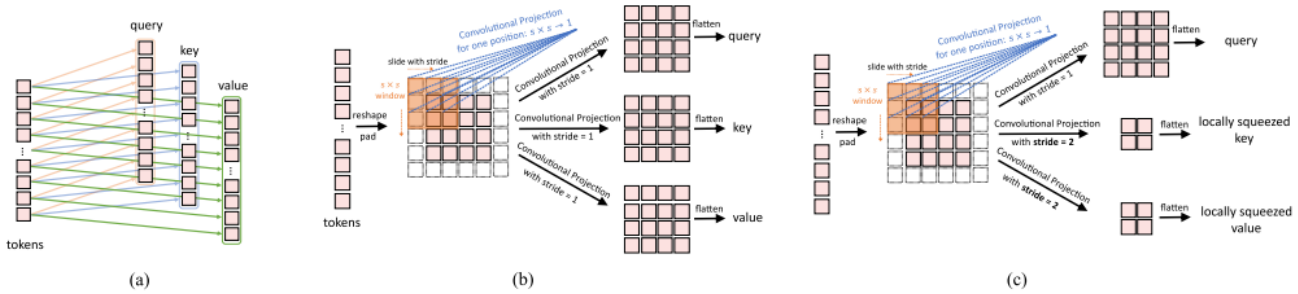
- the classification token is added only in the last stage. Finally, an MLP (i.e. fully connected) Head is utilized upon the classification token of the final stage output to predict the class.



Efficiency considerations - reducing the number of patches in K,V in self attention

The s × s Convolutional Projection permits reducing the number of tokens by using a stride larger than 1. the key and value projection are subsampled by using a convolution with stride larger than 1. We use a stride of 2 for key and value projection, leaving the stride of 1 for query unchanged. In this way, the number of tokens for key and value is reduced 4 times, and the computational cost is reduced by 4 times for the later MHSA operation. This comes with a minimal performance penalty, as neighboring pixels/patches in images tend to have redundancy in appearance/semantics. In addition, the local context modeling of the proposed Convolutional Projection compensates for the loss of information incurred by resolution reduction.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



Twins: Revisiting the Design of Spatial Attention in Vision Transformers

In this work, we surprisingly found that the less favored performance of PVT is mainly due to the absolute positional encodings employed in PVT. As shown in CPVT, the absolute positional encoding encounter difficulties in processing the inputs with varying sizes (which are common in dense prediction tasks). Moreover, this positional encoding also breaks the translation invariance. On the contrary, Swin transformer makes use of the relative positional encodings, which bypasses the above issues. Here, we demonstrate that this is the main cause why Swin outperforms PVT, and we show that if the appropriate positional encodings are used, PVT can actually achieve on par or even better performance than the Swin transformer.

Contributions:

1) Use the conditional position encoding (CPE) proposed in CPVT to replace the absolute PE in PVT. The position encoding generator (PEG) , which generates the CPE, is placed after the first encoder block of each stage. We use the simplest form of PEG, i.e., a 2D depth-wise convolution without batch normalization.

2) Trying to solve the computational complexity in dense prediction tasks - propose the spatially separable self-attention (SSSA) to alleviate this challenge. SSSA is composed of locally-grouped self-attention (LSA) and global sub-sampled attention (GSA).

locally-grouped self-attention (LSA) - equally divide the 2D feature maps into sub-windows, making self-attention communications only happen within each sub-window.

Although the locally-grouped self-attention mechanism is computation friendly, the image is divided into non-overlapping sub-windows. Thus, we need a mechanism to communicate between different sub-windows, as in Swin. Otherwise, the information would be limited to be processed locally, which makes the receptive field small and significantly degrades the performance.

Global sub-sampled attention (GSA). A simple solution is to add extra standard global self-attention layers after each local attention block, which can enable cross-group information exchange. However, this approach is expensive in terms of computation complexity. Here, we use a single representative to summarize the important information for each of m × n sub-windows and the representative is used to communicate with other sub-windows. This is essentially equivalent to using the sub-sampled feature maps as the key in attention

operations, and thus we term it global sub-sampled attention (GSA). The sub-sampling function is regular strided convolution.

Locally shifted attention with early global integration

Motivation

- Due to the expensive quadratic cost of the attention mechanism, either a large patch size is used, resulting in coarse-grained global interactions, or alternatively, attention is applied only on a local region of the image, at the expense of long-range interactions. In this work, we propose an approach that allows for both coarse global interactions and fine-grained local interactions already at early layers of a vision transformer.
- Due to the quadratic cost of the attention mechanism in the number of patches, fixed size partitioning is performed. As a result, ViT does not benefit from the built-in locality bias that is present in CNNs: neighbouring pixels within a patch may be highly correlated, but this bias is not encoded into the ViT architecture. That is, ViT encodes inter-patch correlations well, but not intra-patch correlations. **Further, each image may require a different patch size and location, depending on the size and location of objects in the image.**
- **The method is based on the observation that the optimal location for each patch varies from image to image, depending on object locations and sizes. Therefore, instead of considering a single patch at a given location, we consider an ensemble of patches. This ensemble consists of the conventional fixed patch location and of the patches obtained by small horizontal and vertical shifts of each patch. By employing this shift property, the ensemble can capture more precisely the finer details of the object patches, which are necessary for the downstream task.**

- The method scales well to large images since it does not incur the prohibitive quadratic cost of considering all overlapping patches.
- Dealing with a computational cost:
  To avoid the expensive quadratic cost of computing self-attention over all ensembles of patches, we split each attention layer into two consecutive attention operations, which accumulate both local and global information for each patch. In the local attention layer, we apply self-attention to each patch with its local shifts. In the global attention layers, we utilize the virtually local patches and apply the standard global self-attention between them.

Method

High-level explanation

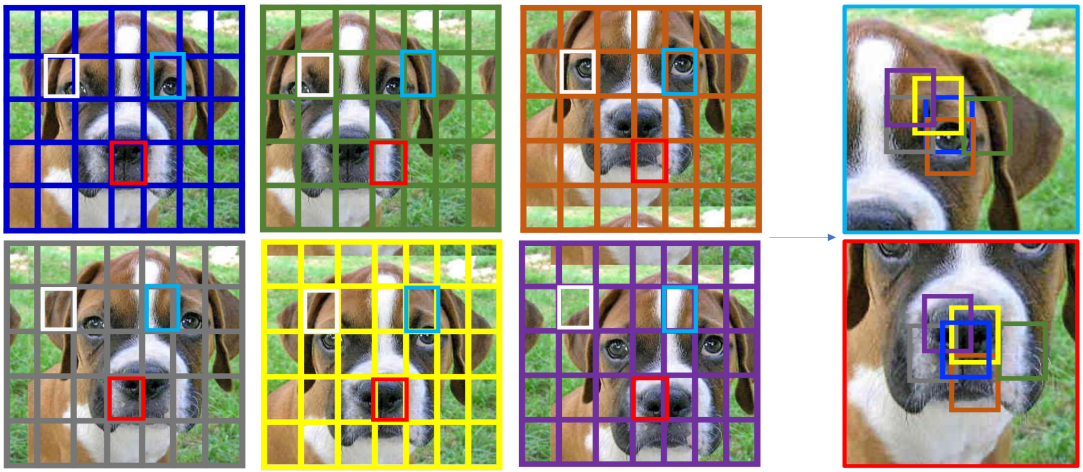
In the local attention layer, we apply attention to each patch and its local shifts, resulting in virtually located local patches, which are not bound to a single, specific location. These virtually located patches are then used in a global attention layer. The separation of the attention layer into local and global counterparts allows for a low computational cost in the number of patches, while still supporting

data-dependent localization already at the first layer, as opposed to the static positioning in other visual transformers.

- **Local attention layer** -
- sum of all possible shifts of the fixed patch.
- **Global attention layer** - utilize the virtually local patches and apply the standard global self-attention between them. This step allows each patch to gain global information from all other patches, where each patch was optimized in the previous local layer by considering all of its local shifts.

Detailed explanation

1) **Local shift embeddings**
   - Given an image, divide it into patches of size SxS resulting in a map with H/S*W/S patches:
     Pass the image through a convolutional layer with out_channels = D and kernel_size=stride =S, to project these patches into hidden features with dimension D.
     The result is a feature map of size $\frac{HW}{S^2} \times D$.
   - Create T-1 variants of this map -
     - Shif the image P_x pixels and then P_y pixels vertically where $P_{x,}, P_y \leq S$ using circular padding at the edges.
     - Each resulting image, referred to as an image shifting variant, is then passed through a convolutional layer with out_channels = D and kernel_size=stride =S.
   - This results in T feature maps of size $\frac{HW}{S^2} \times D$



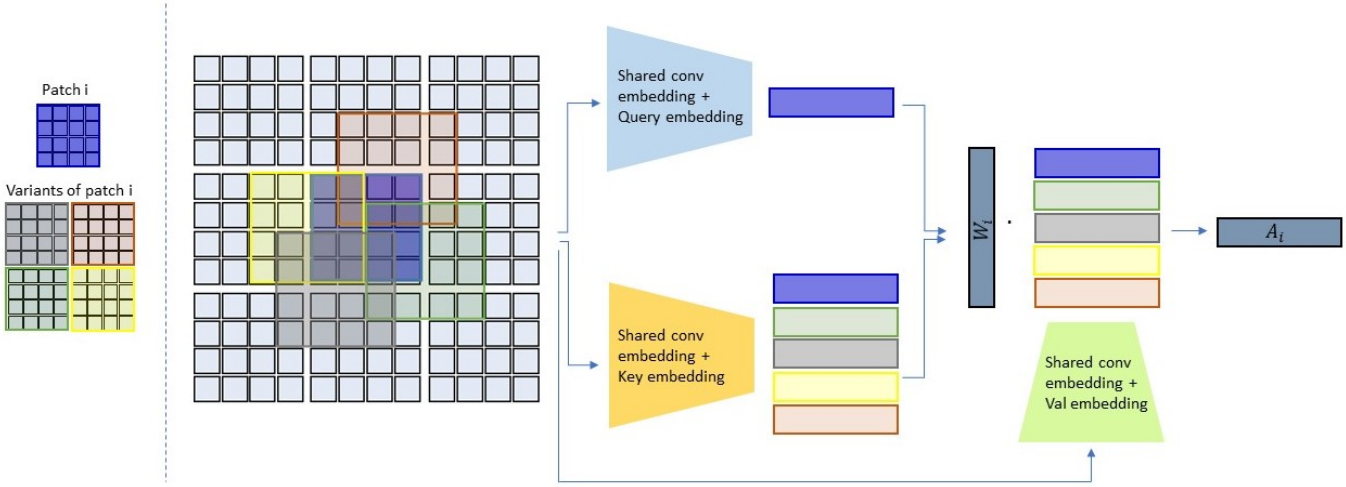
2) **Positional embedding**
   - A unique learned positional embedding is added for each patch embedding in each variant and in each scale. Thus, in each scale, the position embedding is as follow:
     `self.pos_emb = nn.Parameter(torch.zeros(num_variants, num_patches, dim))`
3) **Local attention**
   Denote by N=$\frac{HW}{S^2}$ the number of patches, by X the feature map of all the patches constructed from the original image (without shifting) and by T all the variants feature maps.

- **Calculate the query matrix Q:**
  we use the patches of the non-shifted image as queries:
  $Q = XU_{query}$ $where$ $X \in R^{N \times D}$, $U_{query} \in R^{D \times D}$ $\Rightarrow Q \in R^{N \times D}$
  For technical issues, we unsqueeze it so the overall shape is: $Q \in R^{N \times 1 \times D}$.

- **Calculate keys "K" and values "V":**
  The keys and the values correspond to the shifting variants.
  Demonstration for keys (the same is done for the values):
  $K = ZU_{keys}$ $where$ $Z \in R^{T \times N \times D}$, $U_{keys} \in R^{D \times D}$ $\Rightarrow K \in R^{T \times N \times D}$
  For technical issues, we reshape it to the shape: $K \in R^{N \times T \times D}$**.**
  It means that for each patch (out of N patches), there are T variants for this patch, each variant is represented with a D dimensional embedding.

- **Calculate attention matrix "Attn":**
  Let's look at the shapes:
  $Attn = Q \times K^T$ $\{(N,1,D) \times (N,D,T) = (N,1,T)\} \Rightarrow$ $Attn \in R^{N \times 1 \times T}$
  Now, the attention matrix gives for each patch $i \in [0,..N)$ a socres vector of size T, representing the score of each shifted patch. We now convert it to pseudo-probability vector in dimension T by taking softmax on the last dimension (the variants):
  $Attn = Softmax(Attn, dim = -1) \Rightarrow Attn \in R^{N \times 1 \times T}$
  What happened here? For each "potential" patch in the image, we would like to choose the best location for the patch. Ideally, if there was a "perfect" patch (like the patch that fills the eye of the dog completely) we would give it the maximum score. In practice, for each patch, we get a pseudo-probability vector in dimension T, indicating the weight of the patch variant.

- **Calculate the updated patch embedding:**
  Now, we construct a virtually located patch as a weighted sum of all possible shifts of the fixed patch. Hopefully, the optimal shifts for each image received the highest weight.
  $X' = Attn \times V$ $\{(N,1,T) \times (N,T,D) = (N,1,D)\}$
  We squeeze the resulted tensor and get the output $X' \in R^{N \times D}$



4) **Global Attention**
   Following the local attention block, we are given N tokens (for each patch) with an embedding of size D. So now we can apply the “regular” transformer block on the new feature map.

Optical character recognition

Definition: extracting text from every possible image, be it a standard printed page from a book, or a random image with graffiti in it (“in the wild”).

Traditionally, text recognition has been focussed on document images, where OCR techniques are well suited to digitize planar, paper-based documents. However, when applied to natural scene images, these document OCR techniques fail as they are tuned to the largely black-and-white, line-based environment of printed documents. The text that occurs in natural scene images is hugely variable in appearance and layout, being drawn from a large number of fonts and styles, suffering from inconsistent lighting, occlusions, orientations, noise, and, in addition, the presence of background objects causes spurious false-positive detections. This places text spotting as a separate, far more challenging problem than document OCR.

The majority of methods follow the intuitive process of splitting the task in two: text detection followed by word recognition. Text detection involves generating candidate character or word region detections, while word recognition takes these proposals and infers the words depicted.

Classic-CV approach

- Apply filters to make the characters stand out from the background.
- Apply contour detection to recognize the characters one by one. The contour detection is quite challenging for generalization. it requires a lot of manual fine-tuning, therefore becomes infeasible in most of the problem
- Apply image classification to identify the characters

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Non-Maximum suppression

A typical Object detection pipeline has one component for generating proposals for classification. Proposals are nothing but the candidate regions for the object of interest. Most of the approaches employ a sliding window over the feature map and assign foreground/background scores depending on the features computed in that window. The neighborhood windows have similar scores to some extent and are considered as candidate regions. This leads to hundreds of proposals. As the proposal generation method should have a high recall, we keep loose constraints in this stage. However, processing these many proposals all through the classification network is cumbersome. This leads to a technique that filters the proposals based on some criteria (which we will see soon) called Non-maximum Suppression.

Input: A list of proposal boxes B, corresponding confidence scores S, and overlap threshold N.

Output: A list of filtered proposals D.

Algorithm:

1) Select the proposal with the highest confidence score, remove it from B and add it to the final proposal list D. (Initially D is empty).
2) Now compare this proposal with all the proposals — calculate the IOU (Intersection over Union) of this proposal with every other proposal. For each proposal in B, If the IOU is greater than the threshold N, remove that proposal from B.
3) Again take the proposal with the highest confidence from the remaining proposals in B and remove it from B and add it to D.
4) Once again calculate the IOU of this proposal with all the proposals in B and eliminate the boxes which have high IOU than the threshold.

This process is repeated until there are no more proposals left in B.

Text detection

Key idea: The core of text detection is the design of features to distinguish text from backgrounds.

- Why detecting the whole text and not characters? The performance of the character detector is limited due to three aspects: firstly, characters are susceptible to several conditions, such as blur, non-uniform illumination, low resolution, disconnected stroke, etc.; secondly, a great quantity of elements in the background are similar in appearance to characters, making them extremely hard to distinguish; thirdly, the variation of the character itself, such as fonts, colors, languages, etc., increases the learning difficulty for classifiers.
  By comparison, text blocks possess more distinguishable and stable properties. Both local and global appearances of text block are useful cues for distinguishing between text and non-text regions.

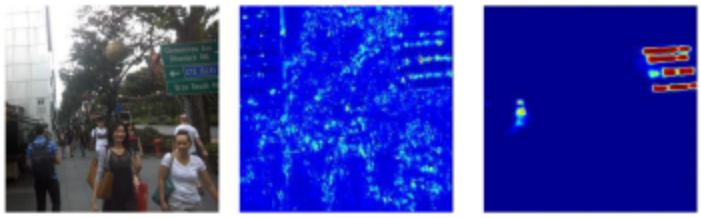


Figure 4. The results of a character detector and our method. (a) An input image; (b) The character response map, which is generated by the state-of-the-art method [8]; (c) The salient map of text regions, which is generated by the Text-Block FCN.

Multi-Oriented Text Detection with Fully Convolutional Networks (2016)

Pipeline:

1) Text blocks are detected using a fully convolutional network (Text-block FCN).

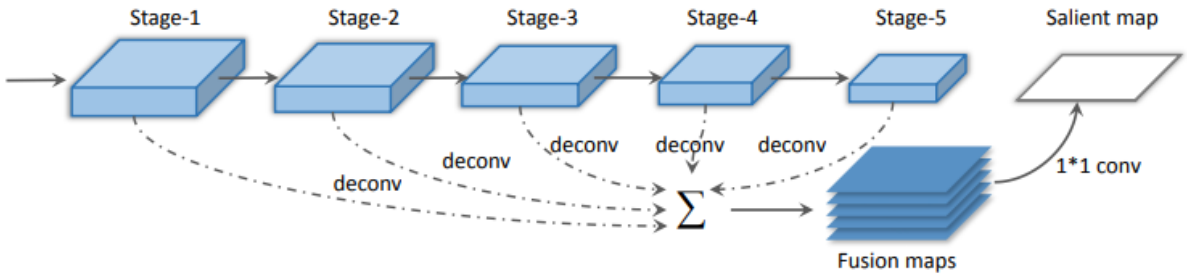


Figure 3. The network architecture of the Text-Block FCN whose 5 convolutional stages are inherited from VGG 16-layer model. For each stage, a deconvolutional layer (equals to a 1×1 convolutional layer and a upsampling layer) is connected. All the feature maps are concatenated with a 1×1 convolutional layer and a sigmoid layer.

Each convolutional stage is followed by a deconvolutinal layer (equals to a 1 × 1 convolutional layer and a upsampling layer) to generate feature maps of the same size. The discriminative and hierarchical fusion maps are then the concatenation in depth of these upsampled maps. Finally, the fully-connected layers are replaced with a 1 × 1 convolutional layer and a sigmoid layer to efficiently make the pixel-level prediction. The feature map os stage 1 captures more local structure while the higher level stages capture more global information.

In the training phase, pixels within the bounding box of each text line or word are considered as the positive region for the following reasons: firstly, the regions between adjacent characters are distinct in contrast to other non-text regions; secondly, the global structure of text can be incorporated into the model; thirdly, bounding boxes of text lines or words are easy to be annotated and obtained.

2) Multi oriented text line candidates are extracted from these text blocks by taking the local information into account.

Although the text blocks detected by the Text-Block FCN provide coarse localizations of text lines, they are still far from satisfactory. To further extract accurate bounding boxes of text lines, taking the information about the orientation and scale of text into account is required. Character components within a text line or word reveal the scale of text. Besides, the orientation of text can be estimated by analyzing the layout of the character components.

- Character Components Extraction - for this they use MSER.
- Orientation estimation

3) False text line candidates are eliminated by the character centroid information. The character centroid information is provided by a smaller fully convolutional network (named Character-Centroid FCN).

EAST: An Efficient and accurate scene text detector (2017)

- A version of the well known U-Net, which is good for detecting features that may vary in size.
- They have experimented with two geometry shapes for text regions, rotated box (RBOX) and quadrangle (QUAD), and designed different loss functions for each geometry. Thresholding is then applied to each predicted region, where the geometries whose scores are over the predefined threshold are considered valid and saved for later non maximum-suppression. Results after NMS are considered the final output of the pipeline.

Pipeline:

1) Fully convolutional network (FCN) that directly produces word or text line level predictions.

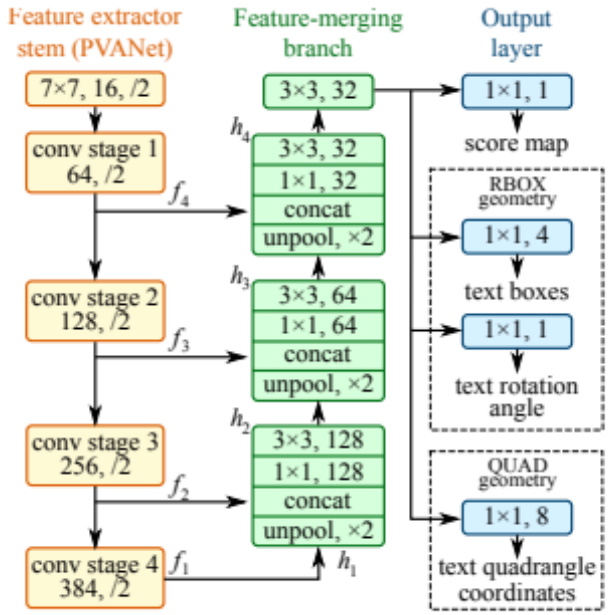


Figure 3. Structure of our text detection FCN.

Several factors must be taken into account when designing neural networks for text detection. Since the sizes of word regions vary tremendously, determining the existence of large words would require features

from the late-stage of a neural network, while predicting accurate geometry enclosing a small word regions need low level information in early stages. Therefore the network must use features from different levels to fulfill these requirements.

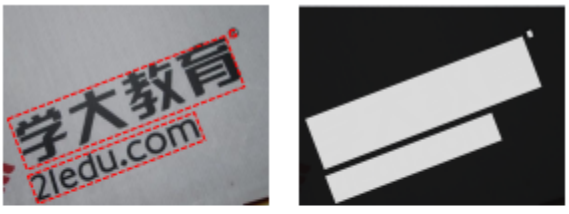
The model can be decomposed into three parts: feature extractor stem, feature-merging branch and output layer. The stem can be a convolutional network pre-trained on ImageNet dataset, with interleaving convolution and pooling layers. Four levels of feature maps with different sizes, denoted as f_i, are extracted from the stem.

In order to merge features maps with different sizes, in each merging stage, the feature map

from the last stage is first fed to an unpooling layer to double its size, and then concatenated with the current feature map. . Next, a conv1×1 bottleneck cuts down the number of channels and reduces computation, followed by a conv3×3 that fuses the information to finally produce the output of this merging stage. Following the last merging stage, a conv3×3 layer produces the final feature map of the merging branch and feeds it to the output layer.

The final output layer contains several conv1×1 operations to project 32 channels of feature maps into 1 channel of score map Fs and a multi-channel geometry map Fg (to Produce RBOX Fg they use convolution with kernel 1x1 and 5 output channels, and for QUAD 8 output channels) .The geometry output can be either one of RBOX or QUAD.

- Explanation of the score map: here, for each pixel the value is probability that it is inside the bounding box, meaning for each pixel it is a part of the text or not (obtaining by sigmoid at the end).It represents the confidence score/level for the predicted geometry map at that location.It lies in the range[0,1].Let's understand it by an example: let's say 0.80 is the score map of a pixel, this simply means that for this pixel we are 80% confident that it will have the predicted geometry map or we can say that it is an 80% chance that the pixel is part of the predicted text region.
  The loss then will be comparing the ground truth score map and the output score map. Example for ground truth:



(a) (b)

Figure 5. The illustration of the ground truth map used in the training phase of the Text-Block FCN. (a) An input image. The text lines within the image are labeled with red bounding boxes; (b) The ground truth map.

For RBOX, the geometry is represented by 4 channels of axis-aligned bounding box R = (AABB) and 1 channel rotation angle θ. The formulation of R is that 4 channels represent 4 distances from the pixel

location to the top, right, bottom, left boundaries of the rectangle respectively. In other words, every pixel has 4 numbers associated with it - distance to each corner of the box.

For QUAD Q, we use 8 numbers to denote the coordinate shift from four corner vertices (4 corners) of the quadrangle to the pixel location. As each distance offset contains two numbers (Δxi , Δyi), the geometry output contains 8 channels.

Loss function: map_score_loss + geometry_loss

- Map score loss - cross-entropy between the GT score map and the predicted score map (instead you can apply the dice loss - very similar to iou. First, multiply the GT map score by the predicted map score and sum. This represents the number of correctly predicted pixels. And then divide by the sum of total pixels of both the GT and the predicted - it is the union of the two bounding boxes. This loss is a measure of overlap between two sets.)
- Geometry loss - regarding the RBOX - we can build a rectangle from the 4 distances of each pixel to the top, bottom, and left. So we can compute Iou between the GT rectangle and the output rectangle/

2) The produced text predictions, which can be either rotated rectangles or quadrangles, are sent to Non-Maximum Suppression to yield final results.

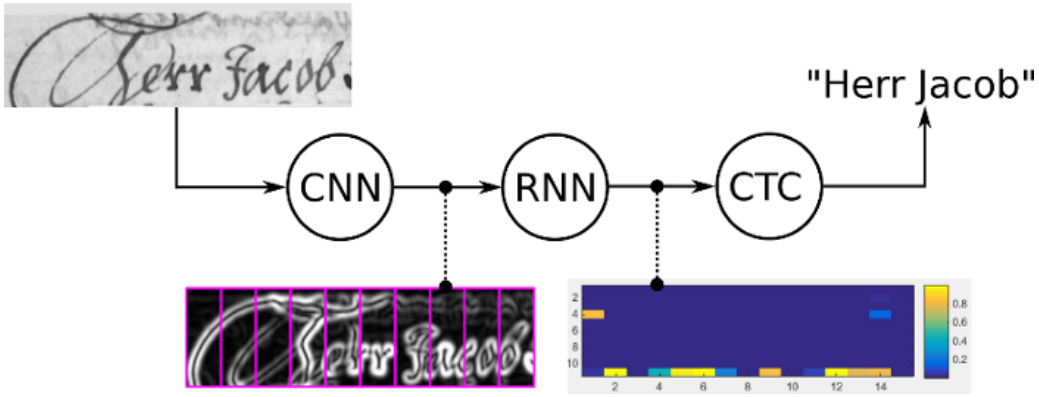
   First, the geometry map is converted back to the boundary boxes. Then we apply thresholding based on the score map to remove some low confidence boxes. (remember that for each pixel, there is a potential box determined by the distances in the geometry map, and also a probability that this pixel is part of the text).
   The remaining boxes are merged using Non-Maximum Suppression.

   Explained: Under the assumption that the geometries from nearby pixels tend to be highly correlated, we proposed to merge the geometries row by row, and while merging geometries in the same row, we will iteratively merge the geometry currently encountered with the last merged one. So for each row, they merge all the boxes that 1) have high overlap with each other, 2) their score is high.

   Weight average of 2 bounding boxes - they merge geometries in the same row, by averaging their coordinates to produce the merged coordinates. Instead of averaging equally, they give a higher weight for each corner in the box with the higher score (from the map score).

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

Text recognition



Text recognition deals with the task of decoding a cropped image that contains one or more words into a digital string of its contents.

Background:

Earlier papers, focused on regular text and used a bottom-up approach, which involved segmenting individual characters with a sliding window, and then recognizing the characters using hand-crafted features. A notable issue with the bottom-up approaches above is that they struggle to use contextual information; instead, they rely on accurate character classifiers.

Recently, most works treat the recognition problem as the sequence prediction problem. Existing methods can be almost divided into two techniques namely Connectionist Temporal Classification (CTC) and attention mechanism. For CTC-based decoding, previous work proposes to use CNN and RNN to encode the sequence features and use CTC for character alignment. For attention-based decoding, proposes recursive CNN to capture longer contextual dependencies and uses an attention-based decoder for sequence generation. One problem is that we can't assume that the text is horizontal, since usually there is text of irregular shapes such as perspective distortion and curvature. To solve the problem of irregular text recognition, several works propose to rectify the text first based on Spatial Transformer Network and then treat it as horizontal text.

Connectionist Temporal Classification (CTC) methods: The NN for such use-cases usually consists of convolutional layers (CNN) to extract a sequence of features and recurrent layers (RNN) to propagate information through this sequence. It outputs character scores for each sequence element, which simply is represented by a matrix. Now, there are two things we want to do with this matrix:

- Train: calculate the loss value to train the NN
- Infer: decode the matrix to get the text contained in the input image

Both tasks are achieved by the CTC operation.

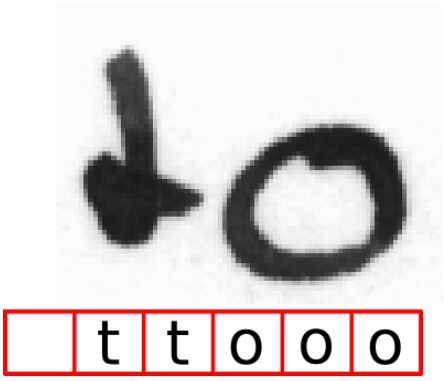
<u>Why do we want to use CTC?</u>

We could, of course, create a data set with images of text lines, and then specify for each horizontal position of the image the corresponding character as shown in the figure below. Then, we could train a NN to output a character score for each horizontal position. However, there are two problems with this Naive solution:

1. it is very time-consuming (and boring) to annotate a data set on character level.

2. we only get character scores and therefore need some further processing to get the final text from it. A single character can span multiple horizontal positions, e.g. we could get " ttooo" because the "o" is a wide character. We have to remove all duplicate "t"s and "o"s. But what if the recognized text would have been "too"? Then removing all duplicate "o"s gets us the wrong result.

How to handle this?



CTC solves both problems for us:

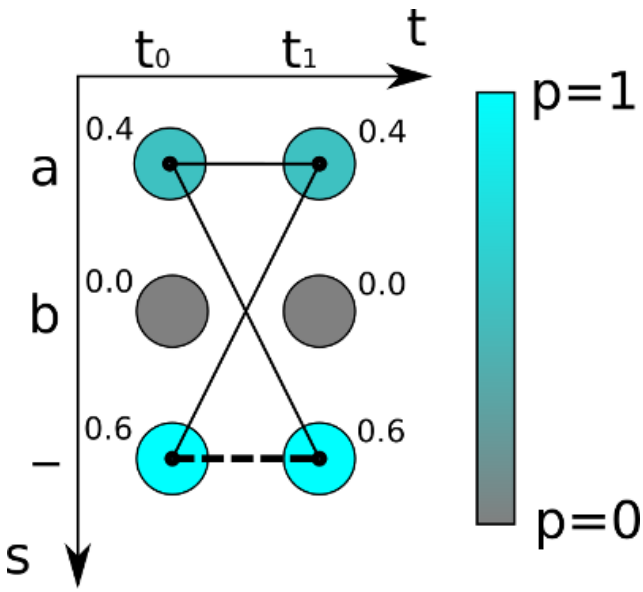
1) we only have to tell the CTC loss function the text that occurs in the image. Therefore we ignore both the position and width of the characters in the image.

2) no further processing of the recognized text is needed.

How does CTC work?

- It tries all possible alignments of the GT text in the image and takes the sum of all scores. This way, the score of a GT text is high if the sum over the alignment scores has a high value.
- All that is left is to compute a score for each alignment - The score for one alignment (or path, as it is often called in the literature) is calculated by multiplying the corresponding character scores together. In the example shown above, the score for the path "aa" is 0.4·0.4=0.16 while it is 0.4·0.6=0.24 for "a-" and 0.6·0.4=0.24 for "-a". To get the score for a given GT text, we sum over the scores of all paths corresponding to this text. Let's assume the GT text is "a" in the example: we have to calculate all possible paths of length 2 (because the matrix has 2 time-steps), which are: "aa", "a-" and "-a". We already calculated the scores for these paths, so we just have to sum over them and get 0.4·0.4+0.4·0.6+0.6·0.4=0.64. If the GT text is assumed to be "", we see that there is only one corresponding path, namely "--", which yields the overall score of 0.6·0.6=0.36.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



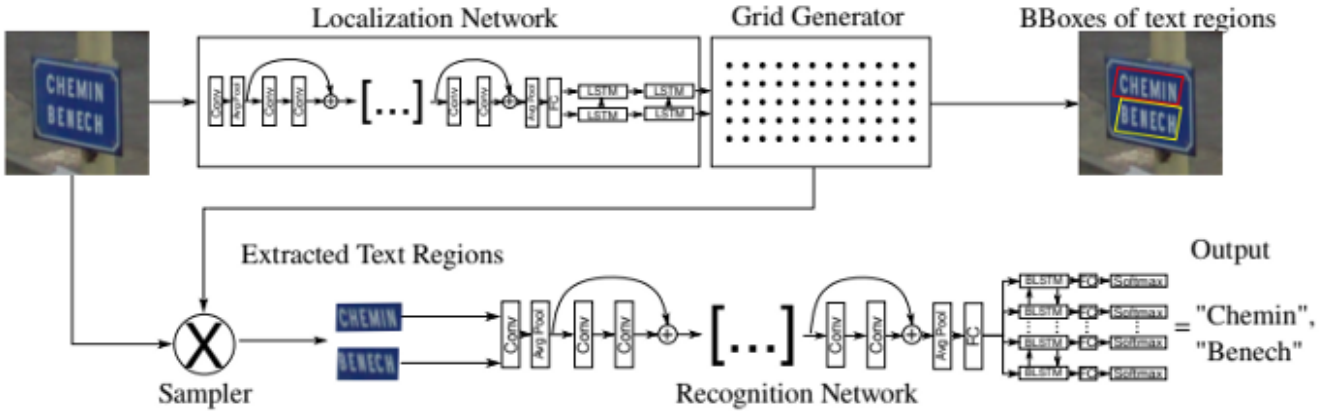
- There was the issue of how to encode duplicate characters. It is solved by introducing a pseudo-character (called blank, but don't confuse it with a "real" blank, i.e. a white-space character). This special character will be denoted as "-" in the following text. We use a clever coding schema to solve the duplicate-character problem: when encoding a text, we can insert arbitrary many blanks at any position, which will be removed when decoding it. However, we must insert a blank between duplicate characters like in "hello". Further, we can repeat each character as often as we like.

Decoding the text:

A simple and very fast algorithm is best path decoding which consists of two steps:

1) Calculate the best path by taking the most likely character per time-step.
2) Undo the encoding by first removing duplicate characters and then removing all blanks from the path. What remains represents the recognized text.

SEE: Towards Semi-Supervised End-to-End Scene Text Recognition (2017)



They train the network with only text annotation (without bounding boxes). This allows them to use more data. The localization network takes the input feature map with C channels, height H and width W and outputs the parameters θ of the transformation that shall be applied. The grid generator is used together with the affine transformation matrices to produce N regular grids. The generated sampling grids are used in two ways: (1) for calculating the bounding boxes of the identified text regions (2) for extracting N text regions.

SCATTER: Selective Context Attentional Scene Text Recognizer (2020)

In this paper, we propose a method for text recognition; we assume the input is a cropped image of text taken from a natural image, and the output is the recognized text string within the cropped image.

Modern methods treat STR (scene text recognition) as a sequence prediction problem. This technique alleviates the need for character-level annotations (per-character bounding box) while achieving superior accuracy. The majority of these sequence-based methods rely on Connectionist Temporal Classification (CTC) or attention-based mechanisms.
Most of the aforementioned STR methods perform a sequential modeling step using a recursive neural network (RNN) or other sequential modeling layers (e.g., multihead attention), usually in the encoder and/or the decoder. This step is performed to convert the visual feature map into a contextual feature map, which better captures long-term dependencies.

Methodology

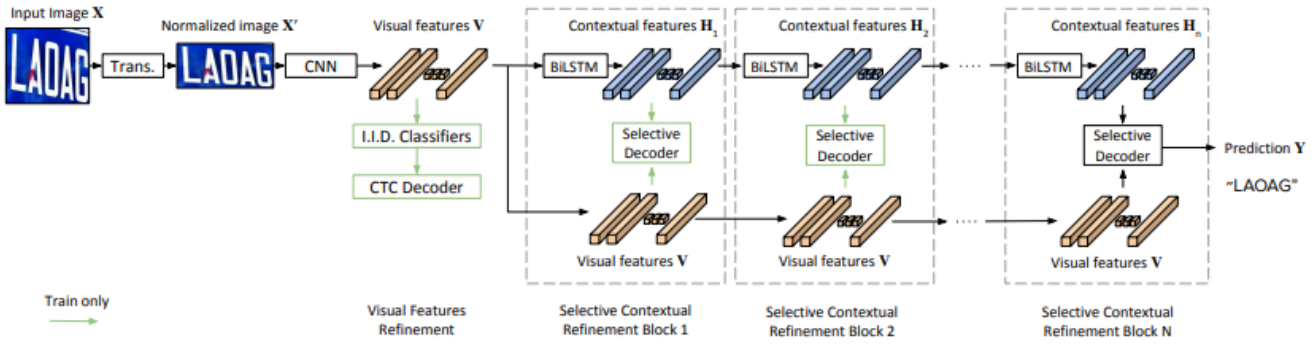


Figure 3: The proposed SCATTER architecture introduces, context refinement, intermediate supervision (additional decoders), and a novel selective-decoder.

The proposed architecture consists of 4 main components:

1) Transformation: the input text image is normalized using a Spatial Transformer Network (STN) .
2) Feature Extraction: maps the input image to a feature map representation while using a text attention module.
   In this step a convolutional neural network (CNN) extracts features from the input image. We use a 29-layer ResNet as the CNN's backbone. The output of the feature encoder is 512 channels by N columns. Specifically, the feature encoder gets an input image X and outputs a feature map F = [f1, f2, ..., fN]. Following the feature map extraction, we use a text attention module. The attentional feature map can be

regarded as a visual feature sequence of length N, denoted as V = [v1, v2, ..., vN], where each column represents a frame in the sequence.
Another point - since each vector i is composed of the same columns in all the feature maps, it means that each vector i is responsible for another region.
To summerize, each feature vector is a descriptor of another region in the image, and we have N descriptors like this.

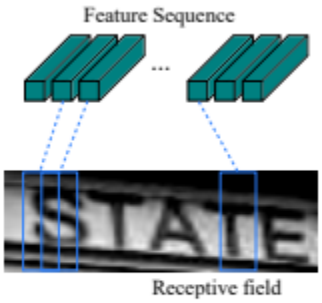


Figure 2. The receptive field. Each vector in the extracted feature sequence is associated with a receptive field on the input image, and can be considered as the feature vector of that field.

3) Visual Feature Refinement: provides direct supervision for each column in the visual features. This part refines the representation in each of the feature columns, by classifying them into individual symbols. Here, they feed V through a fully connected layer that outputs a sequence H of length N. The output sequence is fed into a decoder to generate the final output (with the number of classes).
4) Selective-Contextual Refinement Block: Each block consists of a two-layer BiLSTM encoder that outputs contextual features. The contextual features are concatenated to the visual features computed by the CNN backbone. This concatenated feature map is then fed into the selective-decoder, which employs a two-step 1D attention mechanism. The features extracted by the CNN are limited to its receptive field, and may suffer due to the lack of contextual information. To mitigate this drawback, we employ a two-layer BiLSTM network over the feature map V, outputting H = [h1, h2, ..., hn]. We concatenate the BiLSTM output with the visual feature map, yielding D = (V, H), a new feature space. The feature space D is used both for selective decoding and as an input to the next Selective-Contextual Refinement block.
5) Selective decoder: A selective attention decoder, that simultaneously decodes both visual and contextual features by employing a two-step attention mechanism. The first attention step figures out which visual and contextual features to attend to. The second step treats the features as a sequence and attends the intra-sequence relations.
    - compute attention map on the visual features and the contextual features (concatenation of them) using a fully connected layer.
    - element-wise product is computed between the attention map and D, yielding the attentional features D'.
    - The decoding of D' is done with a separate attention decoder such that for each t-time-step the decoder outputs y_t.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

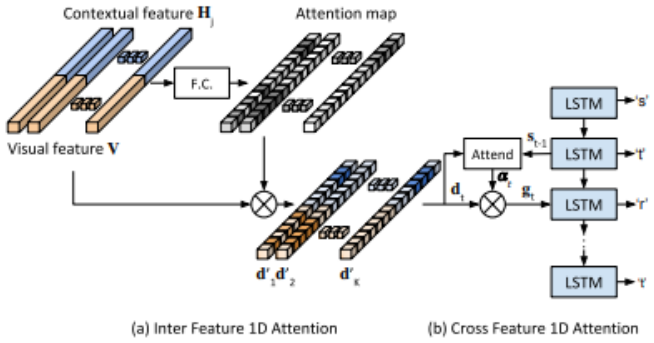


Figure 4: Architecture of the Two-Step Attention Selective-Decoder.

Training loss: the sum of the losses from all the selective contextual refinement blocks where each loss is the negative log-likelihood, and the refined loss of CTC decoder.

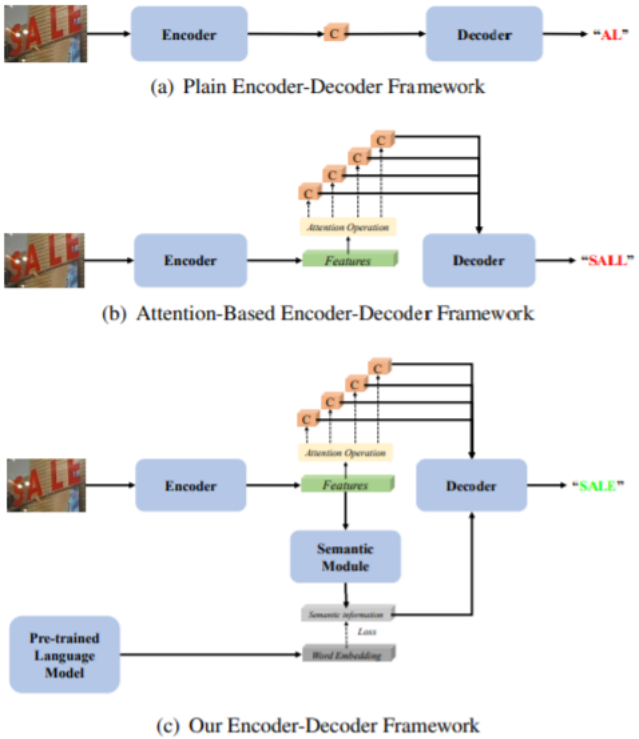
Acceleration in inference: Given a computation budget, improved results can be obtained by first training a deeper network, and then running only the first decoder(s) during inference.

SEED: Semantics Enhanced Encoder-Decoder Framework for Scene Text Recognition (2020)

Motivation: The existing methods define the text recognition task as a sequence character classification task locally, but ignore the global information of the whole word. As a result, they may struggle to handle low-quality images such as image blur, occlusion and incomplete characters. However, people can deal with these low-quality cases well by considering the global information of the text.

Solution: additional semantic information is predicted acting as the global information. The semantic information is then used to initialize the decoder. The semantic information has two main advantages, 1) it can be supervised by a word embedding in natural language processing field, 2) it can reduce the gap between the encoder focusing on the visual feature and the decoder focusing on the language information since the text recognition can be regarded as a cross-modality task

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



Method:

1) The encoder includes CNN backbone and RNN for extracting visual features;
2) The semantic module for predicting semantic information from the visual features;
3) The pre-trained language model for supervising the semantic information predicted by semantic module;
4) The decoder includes RNN with an attention mechanism for generating the recognition results

On Recognizing Texts of Arbitrary Shapes with 2D Self-Attention (2020)

While CNN+RNN methods have brought advances in the field, they are built upon the assumption that input texts are written horizontally. Previous works have collapsed the height component of the 2D image into a 1D feature map (since they assume that text is written horizontally). Realizing the significance and difficulty of recognizing texts of arbitrary shapes, the STR community has put more emphasis on such image types. recent STR approaches are focusing more on addressing texts of irregular shapes.

There are largely two lines of research: (1) input rectification and (2) usage of 2D feature maps.

Input rectification uses spatial transformer net (STN) to normalize text images into canonical shapes: horizontally aligned characters of uniform heights and widths. These methods, however, suffer from the limitation that the possible family of transformations have to be specified beforehand and it is hard to normalize extreme case such as vertically aligned text. Methods using 2D feature maps on the other hand, extract 2D feature maps from input image without collapsing height component and sequentially retrieve characters on the 2D space.

In this paper, we propose an STR network that adopts a 2D self-attention mechanism to capture the spatial dependency in 2D feature map to resolve the remaining challenging case within STR. Our solution, Self-Attention Text Recognition Network (SATRN), adopts the encoder-decoder construct of Transformer to address the

cross-modality between the image input and the text output. The intermediate feature maps are two-dimensional throughout the network. By never collapsing the height dimension, we better preserve the spatial information than prior approaches.

Method:

It consists of an encoder (left column), which embeds an image into a 2D feature map, and a decoder (right column), which then extracts a sequence of characters from the feature map.

Encoder:

The encoder processes input image through a Shallow CNN block that captures local patterns and textures. The feature map is then passed to a stack of self-attention modules, together with an Adaptive 2D positional encoding, a novel positional encoding methodology developed for STR task. The self-attention modules are a modified version of the original Transformer self-attention modules, where the pointwise feed-forward is replaced by our locality-aware feedforward layer.

- Input images are first processed through a shallow CNN that extracts elementary patterns and textures in input images. The CNN performs two convolution layers, each followed by max-pooling layer.
- The feature map produced by the shallow CNN is fed to self-attention blocks. The self-attention block, however, is agnostic to spatial arrangements of its input. positional information plays an important role in recognizing text of arbitrary shape since the self-attention itself is not supplied the absolute location information: given current character location, exactly where in the image can we find the next character? Missing the positional information makes it hard for the model to sequentially track character positions. In STR, it is necessary to adaptively reflect the adjacency along the two directions according to the text alignment in image. For example, in the case of vertically aligned text, the adjacency of the height direction becomes a more important factor than that of width direction in determining the order between characters. On the other hand, for horizontally aligned text, the adjacency in the width direction becomes more important. We thus propose the Adaptive 2D positional encoding (A2DPE) to dynamically determine the ratio between height PE and width PE element depending on the input image.
- For the height and the width of the feature map, they create for each a position embedding. The position embedding for each of the height and width is sinusoidal positional encoding.

  h_encoding = self.get_position_encoding(height, hidden_size, 'h_encoding')

  w_encoding = self.get_position_encoding(width, hidden_size,'w_encoding')

- Now we want to give weight for the h_encoding and the w_encoding (that will give higher weight to the axis where the text align): the weight will be the average pooling of the feature map*learned_weight_h and average pooling of the feature map*learned_weight_w.
  Pos_encoding = alpha*h_encoding + beta*w_encoding.
  Then add the alpha*h_encoding to the features in the H dimension and beta*w_encoding to the features in the width direction.
- Reshape the 2D feature map to be of the shape: batch_size, height*width, hidden_size. Pass it through the self-attention block. The FFN differs from the transformer by capturing better local vicinity around single characters (instead of fully connected they apply convolution)

- Return the features with the shape batch_size, height*width, hidden_size

Decoder:
The decoder retrieves the enriched 2D features from the encoder to generate a sequence of characters. The cross-modality between image input and text output happens at the second multi-head attention module. The module retrieves the next character's visual feature. The feature of the current character is used to retrieve the next character's visual features upon the 2D feature map.

- In training the decoder gets an input the target text shifted one token to the right. (example: instead of dog <eos> it gets <>start dog <eos>.The logic of this is that the output at each position should receive the previous tokens (and not the token at the same position, of course), which is achieved with this shift together with the self-attention mask)
- The position encoding here is the same as in transformer (since we do position encoding to the text0

Read Like Humans: Autonomous, Bidirectional and Iterative Language Modeling for Scene Text Recognition (2021)

1. Vision model

   Background: The attention-based methods follow encoder-decoder architecture, where the encoder processes images and the decoder generates characters by focusing on relevant information from 1D image features.
   The vision model consists of a backbone network and a position attention module. Following the previous methods, ResNet and Transformer units are employed as the feature extraction network and the sequence modeling network. The module of position attention transcribes visual features into character probabilities in parallel, which is based on the query paradigm: Q is positional encodings of character orders, K is the result of applying U-Net on the features, and V are the visual features.

2. Language model

   The BCN is a variant of L-layers transformer decoder. Each layer of BCN is a series of multi-head attention and feed-forward network followed by residual connection and layer normalization.
   The multi-head attention consists of: Q is the positional encodings of character order (or the output from the previous attention block), K, V are character probability obtained from the output of the vision model or the fusion prediction.
   To cope with the problem of noise inputs, we propose iterative LM. The LM is executed M times repeatedly with different assignments for y (the input-ouput of the LM model). For the first iteration, $y_i=1$ is the probability prediction from VM. For the subsequent iterations, $y_i \geq 2$ is the probability prediction from the fusion model in the last iteration. In this way, the LM is able to correct the vision prediction iteratively.

3. Fusion
4. Supervised training
5. Semi-supervised Ensemble Self-training

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

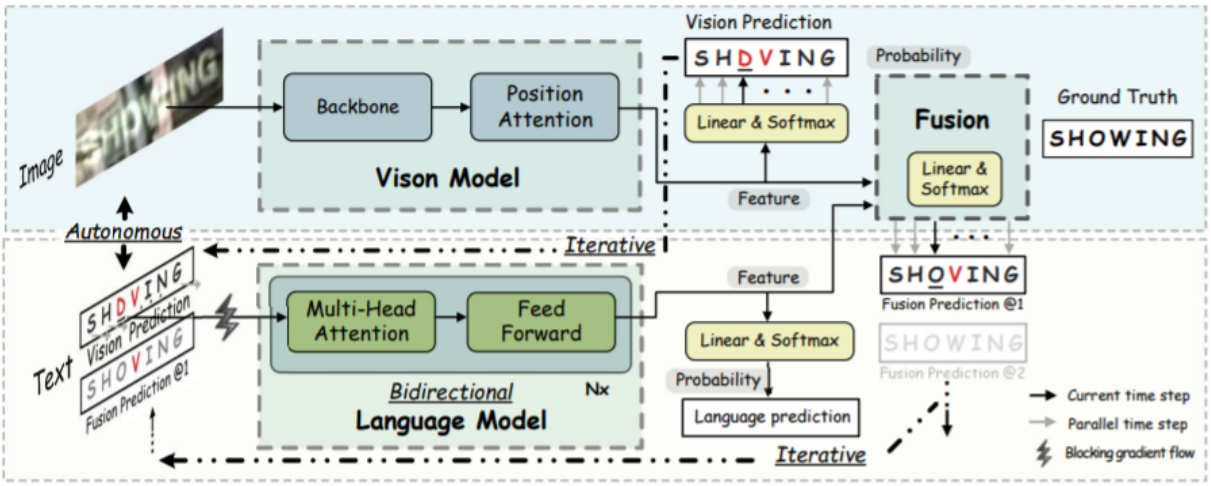


Figure 2. A schematic overview of ABINet.

Text detection + recognition

This task consider first detecting the text in the image and then recognizing the text (classification of the words)

Reading Text in the Wild with Convolutional Neural Networks (2014)

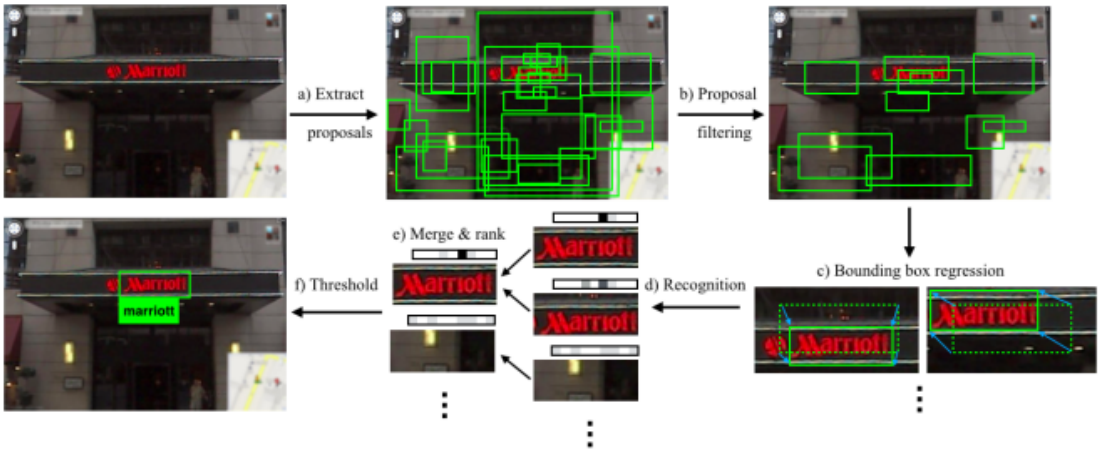
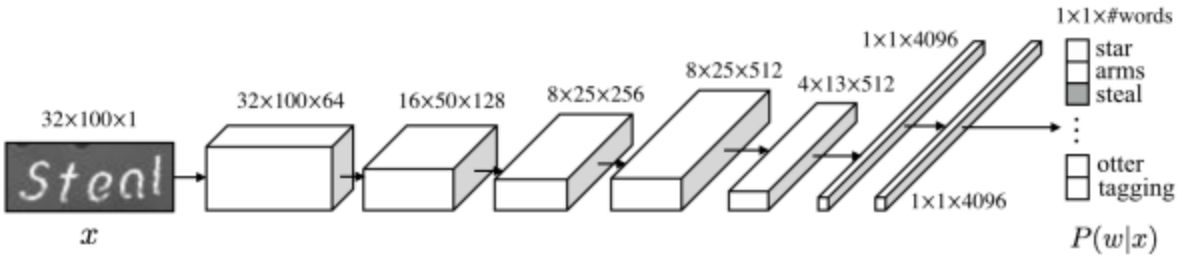


Fig. 1 The end-to-end text spotting pipeline proposed. a) A combination of region proposal methods extracts many word bounding box proposals. b) Proposals are filtered with a random forest classifier reducing number of false-positive detections. c) A CNN is used to perform bounding box regression for refining the proposals. d) A CNN performs text recognition on each of the refined proposals. e) Detections are merged based on proximity and recognition results and assigned a score. f) Thresholding the detections results in the final text spotting result.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)



Pipeline

1) Word bounding box proposal generation
2) Proposal filtering and adjustments - they train a regressor to refine the coordinates of the proposal bounding boxes. A bounding box is parametrized by its top-left and bottom-right corners, such that bounding box b = (x1, y1, x2, y2). The full image I is cropped to a rectangle centered on the region b, with the width and height inflated by a factor of 2 (meaning that the cropped image is bigger by 2 than the bounding box). The resulting image is processed by a CNN to regress the four coordinates of b*. The CNN is trained with example pairs of ground truth box and proposed box. This is done by minimizing the L2 loss between the two boxes.
3) Text recognition - At this stage, a pool of accurate word bounding box proposals have been generated as described previously. We now turn to the task of recognizing words inside these proposed bounding boxes. To this end, we use a deep CNN to perform classification across a pre-defined dictionary of words – dictionary encoding – which explicitly models natural language. The cropped image of each of the proposed bounding boxes is taken as input to the CNN, and the CNN produces a probability distribution over all the words in the dictionary (~90k words in English). The word with the maximum probability can be taken as the recognition result.
4) Final merging for the specific task

Generating synthetic data

1) Font rendering (the word is rendered with random font)
2) Border/shadow rendering, base coloring
3) Projective distortion (projective transformation)
4) Blending with Natural data background
5) Noise

Text generation

Adversarial Generation of Handwritten Text Images Conditioned on Sequences

Starting from a given word, we generate a corresponding image of cursive text. We first tackle the challenge of generating realistic data, and then address the question of using such synthetic data to train neural networks in order to improve the performance of handwritten text recognition.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

The original GAN does not allow any control over the generated images, but many works proposed a modified GAN for class-conditional image generation. However, we want to condition our generation on the sequence of characters to render, not on a single class. One option is to condition the generation on a textual description of the image to be produced. In addition to a random vector, their generator receives an embedding of the description text, and their discriminator is trained to classify as fake a real image with a non-matching description, to enforce the generator to produce description-matching images.

Method:

In order to control the textual content of the generated images, we modify the standard GAN as follows. First, we use a recurrent network (ϕ) to encode s, the sequence of characters to be rendered in an image. G takes this embedding $\phi(s)$ as a second input. Then, the generator is asked to carry out a secondary task. To this end, we introduce an auxiliary network for text recognition (R). We then train G to produce images that R is able to recognize correctly, thereby completing its original adversarial objective with a "collaboration" constraint with R. We use the hinge version of the adversarial loss and the CTC loss to train this system.

ScrabbleGAN: Semi-Supervised Varying Length Handwritten Text Generation

Motivation: This paper deals with handwritten text recognition task. In this case, each author has a unique style, unlike printed text, where the variation is smaller by design. That said, deep learning-based HTR is limited by the number of training examples. Gathering data is a challenging and costly task, and even more so, the labeling task that follows. Thus, the motivation to deal with the small labeled data. One possible approach to reduce the need for data annotation is semi-supervised learning. Semi-supervised methods use, in addition to labeled data, some unlabeled samples to improve performance, compared to fully supervised ones. Consequently, such methods may adapt to unseen images during test time.

Here, they present ScrabbleGAN, a semi-supervised approach to synthesize handwritten text images that are versatile both in style and lexicon. ScrabbleGAN relies on a novel generative model which can generate images of words with an arbitrary length. Generative models (and specifically GANs) are used to synthesize realistic data samples based on real examples. One possible use for these newly generated images is adding them to the original training set, essentially augmenting the set in a bootstrap manner.

Contributions:

1) A novel fully convolutional handwritten text generation architecture
2) The model learns character embeddings without the need for character-level annotation.

Method

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

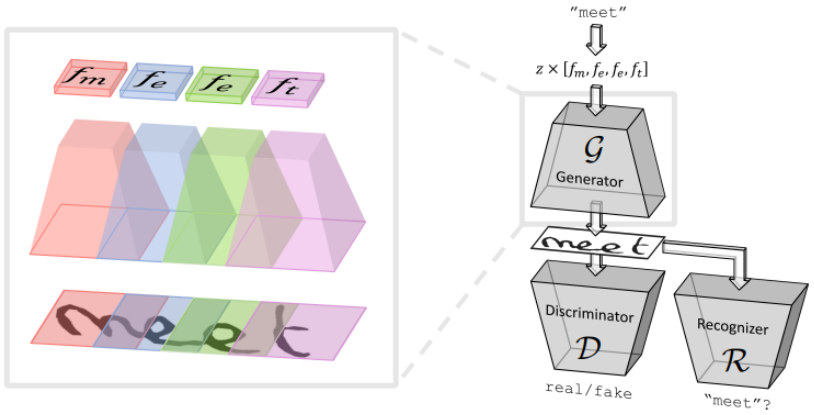


Figure 2: *Architecture overview* for the case of generating the word "meet". **Right**: Illustration of the entire ScrabbleGAN architecture. Four character filters are concatenated (f_e is used twice), multiplied by the noise vector z and fed into the generator $\mathcal{G}$. The resulting image is fed into both the discriminator $\mathcal{D}$ and the recognizer $\mathcal{R}$, respectively promoting style and data fidelity. **Left**: A detailed illustration of the generator network $\mathcal{G}$, showing how the concatenated filters are each fed into a class-conditioned generator, where the resulting receptive fields thereof are overlapping. This overlap allows for adjacent characters to interact, enabling cursive text, for example.

In addition to the discriminator D, the resulting image is alo evaluated by a text recognition network R. while D promotes realistic looking handwriting styles, R encourages the result to be readable and true to the input text.

1) Fully convolutional generator - handwriting is a local process, i.e. when writing each letter is influenced only by its predecessor and successor. The generator is designed to mimic this process: rather than generating the image out of an entire word representation, each character is generated individually, using CNN's property of overlapping receptive fields to account for the influence of nearby letters.
   In other words, the generator can be seen as a concatenation of identical class conditional generators for which each class is a character. Each of these generators produces a patch containing its input character. Each convolutional-upsampling layer widens the receptive field, as well as the overlap between two neighboring characters. This overlap allows adjacent characters to interact, and creates a smooth transition.
   For each character, a filter f_* is selected from a filter-bank F that is as large as the alphabet, for example F = { fa, fb,..., fz} for lowercase English. Four such filters are concatenated in Figure 2 (fe is used twice), and multiplied by a noise vector z, which controls the text style. As can be seen, the region generated from each character filter f_* is of the same size, and adjacent characters' receptive field overlap. This provides flexibility in the actual size and cursive type of the output handwriting character.

2) Style-promoting discriminator - In the GAN paradigm, the purpose of the discriminator D is to tell apart synthetic images generated by G from the real ones. In our proposed architecture, the role of D is also to discriminate between such images based on the handwriting output style. The discriminator architecture has to account for the varying length of the generated image, and therefore is designed to be convolutional as well: The discriminator is essentially a concatenation of separate "real/fake" classifiers with overlapping receptive fields. Since we chose not to rely on character level annotations, we cannot use class supervision for each of these classifiers, as opposed to class conditional GANs. One benefit of this is that we can now use unlabeled images to train D, even from other unseen data corpus. A pooling layer aggregates scores from all classifiers into the final discriminator output.

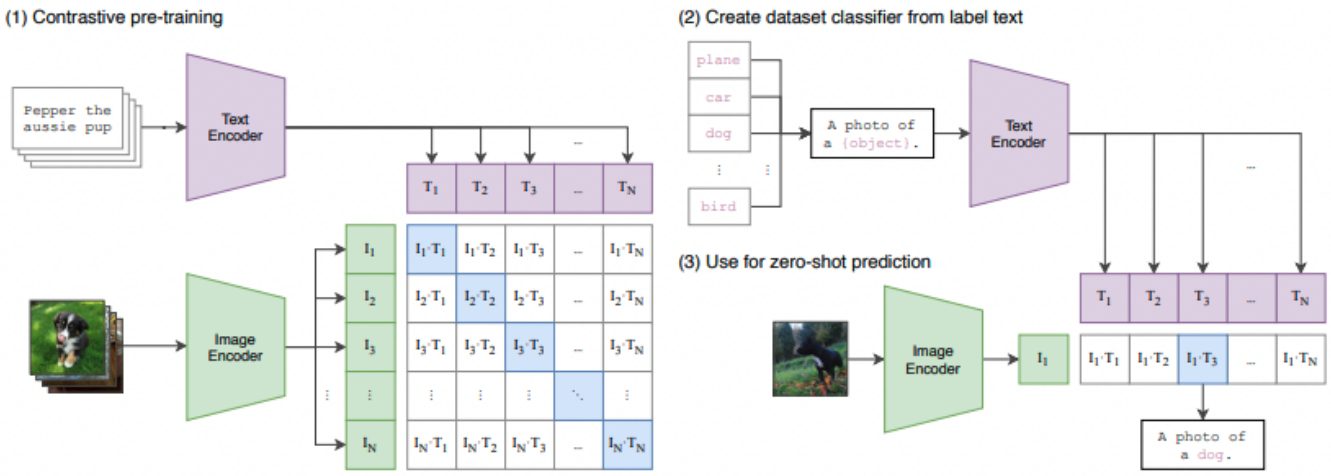
3) Localized text recognizer - While discriminator D promotes real-looking images, the recognizer R promotes readable text, in essence discriminating between gibberish and real text. Generated images are 'penalized' by comparing the recognized text in the output of R to the one that was given as input to G. R is trained only on real, labeled, handwritten samples.

Vision and Language models

Vision and Language tasks include Visual question answering (VQA), Image captioning, Vision-and-Language Navigation, and image-text retrieval. V&L models designed for these tasks often consist of a visual encoder, a text encoder, and a cross-modal interaction module

CLIP

CLIP makes it possible to easily create your own classes for categorization without a need for task-specific training data. CLIP ("Contrastive Language-Image Pre-training") is a neural network that efficiently learns visual concepts from natural language supervision; it is a multi-modal model, trained on a dataset comprising of images and text pairs. The base model uses either a ResNet50 or a Vision Transformer (ViT) architecture as an image encoder and uses a masked self-attention Transformer as a text encoder. These encoders are trained to maximize the similarity of (image, text) pairs via contrastive loss.



Given a batch of N (image, text) pairs, CLIP is trained to predict which of the N × N possible (image, text) pairings across a batch actually occurred. To do this, CLIP learns a multi-modal embedding space by jointly training an image encoder and text encoder to maximize the cosine similarity of the image and text embeddings of the N real pairs in the batch while minimizing the cosine similarity of the embeddings of the $N^2 - N$ incorrect pairings. We optimize a symmetric cross entropy loss over these similarity scores.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

How Much Can CLIP Benefit Vision-and-Language Tasks?

We propose to integrate CLIP with existing V&L models by replacing their visual encoder with CLIP's visual encoder.

- FineTuning - produce text embedding using LSTM/GRU/Masked self-attention Transformer and image embedding using CLIP. Finally, employ an output classifier on top of the fused representation to predict the final answer. We use CLIP visual encoder to extract grid features from the image. For CLIP-ViT-B models, we reshape the patch representation from the final layer into grid features. For CLIP-ResNet, we take the grid features from the last layer before pooling.
- Pretraining - assume a text segment T and an image I as input. As in BERT, the text is embedded as a sequence of word embeddings. The image is embedded as a set of visual vectors from grid-like feature map (extracted from CLIP). The text and visual input are then concatenated into a sequence and processed by a single transformer.

StyleGAN-NADA: CLIP-Guided Domain Adaptation of Image Generators

In this work, we present a text-driven method that enables out-of-domain generation. We introduce a training scheme that shifts the domain of a pre-trained model towards a new domain, using nothing more than a textual

prompt. We encode the difference between domains as a textually-prescribed direction in CLIP's embedding space. We propose a novel loss and a two-generator training architecture. One generator is kept frozen, providing samples from the source domain. The other is optimized to produce images that differ from this source only along a textually-described, crossdomain direction in CLIP-space.

- Directional CLIP loss - Ideally, we want to identify the CLIP-space direction between our source and target domains. Then, we'll fine-tune the generator so that the images it produces differ from the source only along this direction. To do so, we first identify a cross-domain direction In CLIP-space by embedding a pair of textual prompts describing the source and target domains (e.g. "Dog" and "Cat"). We begin with a generator pre-trained on a single source domain (e.g. faces, dogs, churches or cars), and clone it. One copy is kept frozen throughout the training process. Its role is to provide an image in the source domain for every latent code. The second copy is trained. It is fine-tuned to produce images that, for any sampled code, differ from the source generator's only along the textually described direction.

$$\Delta T = E_T\left(t_{target}\right) - E_T\left(t_{source}\right) \ ,$$
$$\Delta I = E_I\left(G_{train}\left(w\right)\right) - E_I\left(G_{frozen}\left(w\right)\right) \ ,$$
$$\mathcal{L}_{direction} = 1 - \frac{\Delta I \cdot \Delta T}{|\Delta I|\,|\Delta T|} \ .$$

  Here, E_I and E_T are CLIP's image and text encoders and t_{target}, t_{source} are the source and target class texts.

- Layer Freezing (for more dramatic changes) - for domain shifts which are predominantly texture based, such as converting photo to sketch, the training scheme described above quickly converges before

overfitting. Ideally, we would like to restrict training to those model weights that are most relevant to a given change. In the case of StyleGAN, it has been shown that codes provided to different network layers influence different semantic attributes. Thus, by considering editing directions in the W+ space – the latent space where each layer of StyleGAN can be provided with a different code wi ∈ W – we can identify which layers are most strongly tied to a given change. Building on this intuition, we suggest a training scheme, where at each iteration we (i) select the k-most relevant layers using the regular CLIP loss, and (ii) perform a single training step where we optimize only these layers, while freezing all others.

DALLE

In this work, we demonstrate that training a 12-billion parameter autoregressive transformer on 250 million image-text pairs collected from the internet results in a flexible, high fidelity generative model of images controllable through natural language.
Our goal is to train a transformer to autoregressively model the text and image tokens as a single stream of data.

- Stage 1: train a VQGAN to compress each 256×256 RGB image into a 32 × 32 grid of image tokens, each element of which can assume 8192 possible values.
- Stage 2: concatenate the BPE-encoded text tokens with the image tokens, and train an autoregressive transformer to model the joint distribution over the text and image tokens.

GLIDE: Towards Photorealistic Image Generation and Editing with Text-Guided Diffusion Models

Recent text-conditional image models are capable of synthesizing images from free-form text prompts, and can compose unrelated objects in semantically plausible ways. However, they are not yet able to generate photorealistic images that capture all aspects of their corresponding text prompts.
On the other hand, unconditional image models can synthesize photorealistic images, sometimes with enough fidelity that humans can't distinguish them from real images. Within this line of research, diffusion models have emerged as a promising family of generative models, achieving state-of-the-art sample quality on a number of image generation benchmarks.
Motivated by the ability of guided diffusion models to generate photorealistic samples and the ability of text-to-image models to handle free-form prompts, we apply guided diffusion to the problem of text-conditional image synthesis.
First, we train a 3.5 billion parameter diffusion model that uses a text encoder to condition on natural language descriptions. Next, we compare two techniques for guiding diffusion models towards text prompts: CLIP guidance and classifier-free guidance.

- Classifier free guidance

  A technique for guiding diffusion models that does not require a separate classifier model to be trained. For classifier-free guidance, the label y in a class-conditional diffusion model θ(xt|y) is replaced with a null label ∅ with a fixed probability during training. During sampling, the output of the model is extrapolated further in the direction of θ(xt|y) and away from θ(xt|∅) as follows:

$$\hat{\epsilon}_\theta(x_t|y) = \epsilon_\theta(x_t|\emptyset) + s \cdot (\epsilon_\theta(x_t|y) - \epsilon_\theta(x_t|\emptyset))$$

To implement classifier-free guidance with generic text prompts, we sometimes replace text captions with an empty sequence (which we also refer to as ∅) during training. We then guide towards the caption c using the modified prediction $\hat{\varepsilon}$:

$$\hat{\epsilon}_\theta(x_t|c) = \epsilon_\theta(x_t|\emptyset) + s \cdot (\epsilon_\theta(x_t|c) - \epsilon_\theta(x_t|\emptyset))$$

At the first stage, train the text-to-image model (base model) regularly. After the initial training run, fine-tune the base model to support unconditional image generation. This training procedure is exactly like pre-training, except 20% of text token sequences are replaced with the empty sequence. This way, the model retains its ability to generate text-conditional outputs, but can also generate images unconditionally.

- CLIP guidance

  Since CLIP provides a score of how close an image is to a caption, several works have used it to steer generative models like GANs towards a user-defined text caption. To apply the same idea to diffusion models, we can replace the classifier with a CLIP model in classifier guidance. In particular, we perturb the reverse-process mean with the gradient of the dot product of the image and caption encodings with respect to the image:

$$\hat{\mu}_\theta(x_t|c) = \mu_\theta(x_t|c) + s \cdot \Sigma_\theta(x_t|c)\nabla_{x_t}\left(f(x_t) \cdot g(c)\right)$$

Training

- Train a 3.5 billion parameter text-conditional diffusion model at 64 × 64 resolution - For each noised image x_t and corresponding caption c, the model predicts $p(x_{t-1}|x_t,\ c)$. To condition on the text, we first encode it into a sequence of K tokens, and feed these tokens into a transformer model. The final token embedding of the transformer is used as a class embedding in the diffusion model.
- Train a 1.5 billion parameter text-conditional upsampling diffusion model to increase the resolution to 256 × 256. This model is conditioned on text in the same way as the base model, but uses a smaller text encoder.
- For CLIP guidance, we also train a noised 64 × 64 ViT-L CLIP model - for this, they train a noised CLIP model with image encoder that receives noise images.

Image inpainting

We explicitly fine-tune our model to perform inpainting/During fine-tuning, random regions of training examples are erased, and the remaining portions are fed into the model along with a mask channel as additional conditioning information. We modify the model architecture to have four additional input channels: a second set of RGB channels, and a mask channel. We initialize the corresponding input weights for these new channels to zero before fine-tuning. For the upsampling model, we always provide the full low resolution image, but only provide the unmasked region of the high-resolution image.

Written by Shelly Sheynin (shelly.sheynin3@gmail.com)

SLIP: Self-supervision meets Language-Image Pre-training

A framework for combining language supervision and image self-supervision to learn visual representations without category labels. During pre-training, separate views of each input image are constructed for the language supervision and image self-supervision branches, then fed through a shared image encoder.

During each forward pass in SLIP, all images are fed through the same encoder. Each image is augmented to two views for SimCLR, and cropped for CLIP. Then, all the 3 images are encoded using image encoder (ViT), and the text is encoded using text encoder (the text transformer from CLIP). Then, for the SSL objective, the 2 views encodings are projected using 3 layers MLP, and for the CLIP objective, the image encoding+text encoding are embedded using separate linear projections. The CLIP and SSL objectives are computed on the relevant embeddings and then summed together into a single scalar loss.