0.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) were originally developed as a mathematical model of the biological brain (**McCulloch1943**; **Rosenblatt58theperceptron**; **Rumelhart1987**). Although ANNs have little resemblance to real biological neurons, they are a powerful Machine Learning (ML) tool and one of the most popular research topics in the last years. Nowadays, most researchers have shifted from the prespective of the biological neuron model to a more general *function approximator* point of view; in fact, it was proved that ANNs with enough capacity are capable of approximating any measurable function to any desired degree of accuracy (**Cybenko1989**; **Hornik1991251**); this is, however, a non-constructive proof.

The basic structure of ANNs is a network of nodes (usually called neurons) joined to each other by weighted connections. Many varieties of ANNs have appeared over the years with different properties. One important distinction is between ANNs whose connections form feedback loops, and those whose connections are acyclic. ANNs with cycles are typically referred to as recurrent neural networks and those without cycles are known as Feedforward Neural Networks (FNNs).

In this work, only FNNs are used, in particular, a special kind that makes use of the convolution operation called Convolutional Neural Networks (CNNs). The following section provides an overview of this networks as well as the basic principles of training them.

0.1.1 Convolutional Neural Networks (CNN)

CNNs are a kind of ANNs particulary well-suited for computer vision tasks. They were first introduced in **LeCun1998** to perform the task of hand-written digit classification and later popularized by **Krizhevsky2012** entry on the ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012)[1], which won the classification task with a large margin of 10%.

CNNs consist of an input and an output layer, as well as multiple hidden layers (see figure 0.1). The input layer contains the data (e.g., RGB image) with minimal preprocessing (normalization, cropping...), in contrast to other ML algorithms that need hand-engineered features. The output layer is different depending on the task.

Each hidden layer performs the convolution operation with one or more filters (com-

[1]ILSVRC is a competition to estimate the content of photographs for the purpose of retrieval and automatic annotation using a subset of the large hand-labeled ImageNet dataset (around 10,000,000 labeled images depicting 10,000+ object categories) as training. In the classification task, the algorithms are evaluated by the error rate in the test images presented with no initial annotation.

monly refered to as kernels by the Deep Learning (DL) community) taking the previous layer's output as the input and then an element-wise non-linear function is applied to the output. The non-linearities allow the model to extract hierarchical features (early layers extract the called low-level features and deeper layers extract high-level features) from the input data as it is illustrated in figure 0.2 on page 8 extracted from **Zeiler2014**.

Down-sampling (also known as pooling in the DL literature) is also a very common operation applied after some hidden layers, aimed to make the model tranlation-invariant and reduce memory needs. Three main methods can be used to represent the set of N (or $N \times N$) neighbouring samples with a single number: *a*) Max-pooling uses the maximum value; *b*) The average value is used by the average-pooling method and *c*) Standard decimation "takes" a sample out of every N samples, it is usually implemented via N-strided (skipping $N-1$ positions when sliding the filter) convolution to compute only the used values.

Other kinds of layers like dropout (**Srivastava2014**) and batch-normalization (**Ioffe2015**) can be used for regularization or faster training process.

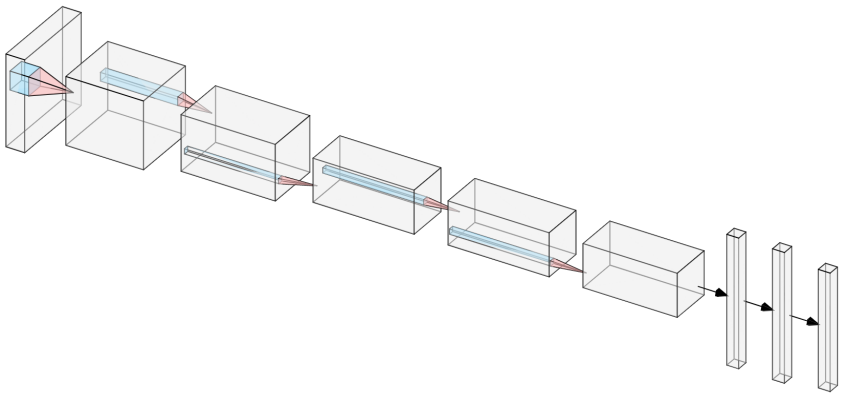

Figure 0.1: Visualization of a CNN with five convolutional layers and two fully-connected layers (a kind of layer not described in this work).

0.1.2 Training ANNs

Finding (or learning) the best set of parameters (θ) (weights, filters, ...) of a network for a given problem can be posed as an optimization problem by defining an appropiate objective function ($\mathcal{J}$). In a supervised setting, where training data composed by inputs ($\mathbf{X}$) and targets ($\mathbf{Y}$) is available:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, \mathcal{J}\left((\mathbf{X}, \mathbf{Y}), f_\theta\right) \tag{0.1}$$

The complex structure of ANNs makes this optimization problem non-convex, hence iterative methods are used for moving towards an optimum solution; particulary, gradient

descent based are the most common type as the networks are —by construction— fully-differentiable.

The idea behind gradient descent is simple. Given a (random) initial value for the input variables (e.g., filter coefficients in the case of CNNs), these are updated by moving towards the direction with greater slope i.e., the gradient. Moving towards the direction of the gradient (gradient ascent) will yield a local maximum, useful when maximizing the objective function; while moving toward the direction oposite to the gradient will yield a local minimum:

$$\theta_i^{(n)} \leftarrow \theta_i^{(n-1)} - \mu \frac{\partial \mathcal{J}}{\partial \theta_i}|_{\theta_i^{(n-1)}} \tag{0.2}$$

where the superscript (n) denotes the n-th iteration step and μ (known as learning rate) controls how much the variables should change between steps. Note that the length of the "step" is not only governed by the learning rate but also by the magnitute of the gradient.

In a Deep Neural Network (DNN) with thousands or millions of parameters, computing the gradient with respect to each parameter independently is computationally restrictive. In **backprop** work, an algorithm to efficiently train ANNs called backpropagation[2] which uses the principles of dynamic programming and exploits the chain rule was presented and it is how almost all ANNs are trained nowadays.

Cost function When training ANNs, the objective function is commonly defined as a cost function with two parts: the expected value of a loss plus a (weighted) regularization term.

$$\mathcal{J}\left((\mathbf{X}, \mathbf{Y}), f_\theta\right) = \mathbb{E}\{L\left(f_\theta(\mathbf{X}), \mathbf{Y}\right)\} + \lambda R(f_\theta) \tag{0.3}$$

$$\approx \frac{1}{N} \sum_{\mathbf{x}_i, \mathbf{y}_j \in \mathbf{X}, \mathbf{Y}} L\left(f_\theta(\mathbf{x}_i), \mathbf{y}_j\right) + \lambda R(f_\theta) \tag{0.3a}$$

Equation (0.3a) shows how the expected value is approximated by taking the mean over N samples (batch size) of the dataset. Stochastic gradient descent is the case where $N = 1$, and mini-batch gradient descent when N is smaller than the total number of samples in the dataset; a small batch size is almost mandatory for large datasets, as computing the loss for every sample at each iteration step is very time and memory expensive and not only that: a small batch size helps avoiding bad local optima and improve generalization (**DBLP:journals/corr/abs-1804-07612**; **Zhang2017No0**).

[2]This lead to the terminology of forward pass, when the output of the model is computed sequentially from the input layer through each of the hidden layers, and the backward pass, when the partial derivatives a computed starting from the final layer and going back to the first hidden layer.

The loss function will depend on our task. For example, in classification problems with c clases the cross entropy can be used: $L(\hat{\mathbf{y}}_i, \mathrm{y}_i) = -\log[\hat{\mathbf{y}}_i]_{\mathrm{y}_i}, \quad \mathrm{y}_i \in \{0 \,..\, c-1\}, \hat{\mathbf{y}}_i \in \mathbb{R}^c$; this loss enforces the model to make a good estimation of the class probabilities given an input datapoint. A loss function well aligned with our task is crucial, but defining such mathematical description of some problems is not always straightforward.

Advanced gradient-based optimization methods The basic (stochastic/mini-batch) gradient descent methods tend to find bad sub-optima when dealing with the noisy, non-convex landscape of ANNs; with a performance very sensitive to the initial values, learning rate and batch size. Many research work (**Adagrad**; **Hinton2012RMSProp**; **Zeiler2012ADADELTA**; **Kingma2014Adam**) has focused on this area, developing algorithms that try to find better optima with fewer iterations. The one used in this work is the Adam "optimizer" (**Kingma2014Adam**) which defines the following update rule based on adaptive estimates of gradient moments:

$$\theta_i^{(n)} \leftarrow \theta_i^{(n-1)} - \mu \frac{\hat{m}_i^{(n)}}{\sqrt{\hat{v}_i^{(n)}} + \epsilon} \tag{0.4}$$

$$\hat{m}^{(n)} \leftarrow \frac{m^{(n)}}{1-\beta_1^n}, \quad m_i^{(n)} \leftarrow \beta_1 m_i^{(n-1)} + (1-\beta_1) g_i^{(n)} \tag{0.4a}$$

$$\hat{v}_i^{(n)} \leftarrow \frac{v_i^{(n)}}{1-\beta_2^n}, \quad v_i^{(n)} \leftarrow \beta_2 v_i^{(n-1)} + (1-\beta_2) \left(g_i^{(n)}\right)^2 \tag{0.4b}$$

$$g_i^{(n)} \leftarrow \frac{\partial \mathcal{J}}{\partial \theta_i}|_{\theta_i^{(n-1)}} \tag{0.4c}$$

The parameter update equation (0.4) looks similar to the one in (0.2) but instead of directly using the gradient, its based on the exponential moving average of the gradient ($\hat{m}$) with a coefficient of $1-\beta_1$ (equation 0.4a) and the exponential moving average of the squared gradient ($\hat{v}$) with a coefficient of $1-\beta_2$ (equation 0.4b). The β's are refered to as momentum an typical values lie around 0.9. ϵ is a very small constant (e.g, 10^{-8}) to avoid division by zero. The intuition behind this update rule is that the steps are forced to be of size μ, as we are dividing by the magnitude of the gradient; and instead of taking the direction of the gradient at a given iteration step, the gradient value is passed through a low-pass filter to avoid making sudden changes that are common in stochatic optimization (specially with a small batch size).

0.2 Generative Adversarial Networks (GANs)

In ML generative models are ones which model the distribution of the data. This models can be used to perform classification through the conditional distribution (via a prior distribution of the classes) or to sample/generate data. Multiple generative models have been proposed that make use of DNN (**Hinton2009**; **Kingma2013**; **Goodfellow2014**), Generative Adversarial Networks (GANs) in particular have gained a lot of attention as they are capable of generating realistic images.

GANs is a framework for estimating generative models via an adversarial process, in which two models are trained: a generative model (G) that captures the data distribution, and a discriminative model (D).

Both models are implemented as DNN (usually CNNs in the case of image data). $G_{\theta_g}(\mathbf{z})$ maps from an input space of noise variables with distribution $p_z(\mathbf{z})$ to data space. $D_{\theta_d}(\mathbf{x})$ outputs the probability that $\mathbf{x}$ came from the data distribution $p_{data}(\mathbf{x})$ rather than G. See figure 0.3 for a simple illustration.

G is trained to minimize the likelihood of D assigning a low probability to its samples. While D is simultaneously trained to maximize the probability of assigning the correct label to both training examples and samples from G. Hence, the loss functions (see equation (0.3)) for each model are defined as (note how they go one against the other):

$$L_G = \log(1 - D(G(\mathbf{z}))) \tag{0.5a}$$

$$L_D = \begin{cases} -\log(D(\mathbf{x}), & \mathbf{x} \sim p_{data} \\ -\log(1 - D(\mathbf{x})) \equiv -\log(1 - D(G(\mathbf{z}))), & \mathbf{x} \sim p_g \end{cases} \tag{0.5b}$$

Equivalently, D and G play the following two-player minimax game:

$$\min_G \max_D \mathbb{E}_{x \sim p_{data}(x)}\{\log D(x)\} + \mathbb{E}_{z \sim p_z(z)}\{\log(1 - D(G(z)))\} \tag{0.6}$$

If G and D have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{data}$. The discriminator will be unable to differentiate between the two distributions, i.e. $D(x) = 0.5$.

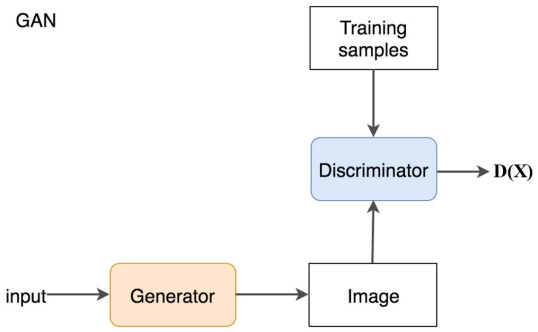

Figure 0.3: GANs training process diagram

This way of traning a generative model is unstable and can fail to converge due to a number of problems known as *failure modes*:

- Mode collapse: the generator collapses which produces limited varieties of samples,
- Diminished gradient: the discriminator gets too successful that the generator gradient vanishes and learns nothing,
- Unbalance between the generator and discriminator causing overfitting,

A number of publications (**Arjovsky2017**; **miyato2018spectral**; **DBLP:journals/corr/Salim** tackle this problems with architecture and loss function modifications, and practical techniques. Successfuly tranined generators, generate very realistic samples as the constructed objective function essentially says "generate samples that look realistic".

0.2.1 GANs for image-to-image translation

In this section, two frameworks for image-to-image translation based on GANs are described. The first one, called pix2pix, (**Isola2016**) uses a conditional generative adversarial network to learn a mapping from input to output images using paired data. On the other hand, Cycle-Consistent Generative Adversarial Networks (CycleGANs) (**Zhu2017a**) mapping is learned without paired data. Paired data consists of training examples $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^{N}$ where correspondence between $\mathbf{x}_i$ and $\mathbf{y}_i$ exists, obtaining this kind of data can be difficult (or impossible) and expensive; contrarily, unpaired data consists of a source set $\mathbf{X}$ and a target set $\mathbf{Y}$ with no information provided as to which $\mathbf{x}_i$ matches which $\mathbf{y}_i$ (if any).

Pix2Pix **Isola2016** work presents a conditional adversarial setting and applies it successfully to a variety of image-to-image translation problems that traditionally would

require very different loss formulations; proving that *learned loss functions* are versatile. The losses used are similar to the ones in equations (0.5a) and (0.5b); but the discriminator not only gets the output of the generator as input, but the corresponing input as well (see figure 0.4 extracted from **Isola2016**). Apart from that, the generator is tasked to also be near the ground truth in a L1 sense; this is done by modifying the generator's loss function:

$$L_G = \log(1 - D(\mathbf{x}, G(\mathbf{x}))) + \lambda \left\| \mathbf{y} - G(\mathbf{x}) \right\|_1 \tag{0.7}$$

CycleGAN **Zhu2017a** presents a method that builds on top of the Pix2Pix framework for capturing special characteristics of one image collection and transfering these into another image collection in the absence of any paired training examples. In theory, an adversarially trained generator can learn to map images from a domain X to look indistiguishable from images from a domain Y; in practice, it is difficult to optimize the adversarial objective in isolation as this often leads to the mode collapse problem. In **Zhu2017a** work, this problem is addressed by adding more structure to the objective; concretely, it "encourages" the mapping to be cycle-consistent, i.e.: a function $G_{X \to Y}$ that maps from domain X to Y should have and inverse $G_{Y \to X}$ that maps its output to the original input; as in language translation, if a sentence is tranlated from Spanish to English and then back to Spanish we should arrive to a sentence close to the original. $G_{Y \to X}(G_{X \to Y}(\mathbf{x})) \approx \mathbf{x}$. This is done by simultaneously training two generators (with corresponding discriminators D_X, D_Y —notice how in this case, these do not take the source image as input—) and tasking them to not only "fool" their discriminator but to also produce an image that is close to the original input when translated back to the source domain (using the complementary generator), this is done by defining the following losses for the generators (more clearly visualized in figure 0.5).
An aditional term called idenity loss can be added to encourage the mapping to preserve color composition between the input and the output by making the generator be near an idenity mapping when samples from the target domain are provided. Note that the input and output domains need to have the same number of channels.
The final losses for the generators are the following:

$$\begin{aligned} L_{G_{X \to Y}} = & \log(1 - D_Y(G_{X \to Y}(\mathbf{x}))) \\ & + \lambda_{cycle} \left\| \mathbf{x} - G_{Y \to X}(G_{X \to Y}(\mathbf{x})) \right\|_1 \\ & + \lambda_{identity} \left\| \mathbf{y} - G_{X \to Y}(\mathbf{y}) \right\|_1 \end{aligned} \tag{0.8a}$$

$$\begin{aligned} L_{G_{Y \to X}} = & \log(1 - D_X(G_{Y \to X}(\mathbf{y}))) \\ & + \lambda_{cycle} \left\| \mathbf{y} - G_{X \to Y}(G_{Y \to X}(\mathbf{y})) \right\|_1 \\ & + \lambda_{identity} \left\| \mathbf{x} - G_{Y \to X}(\mathbf{x}) \right\|_1 \end{aligned} \tag{0.8b}$$

Note that neither Pix2Pix nor CycleGAN generators use a noise distribution to generate samples (in contrast to the original GANs framework).

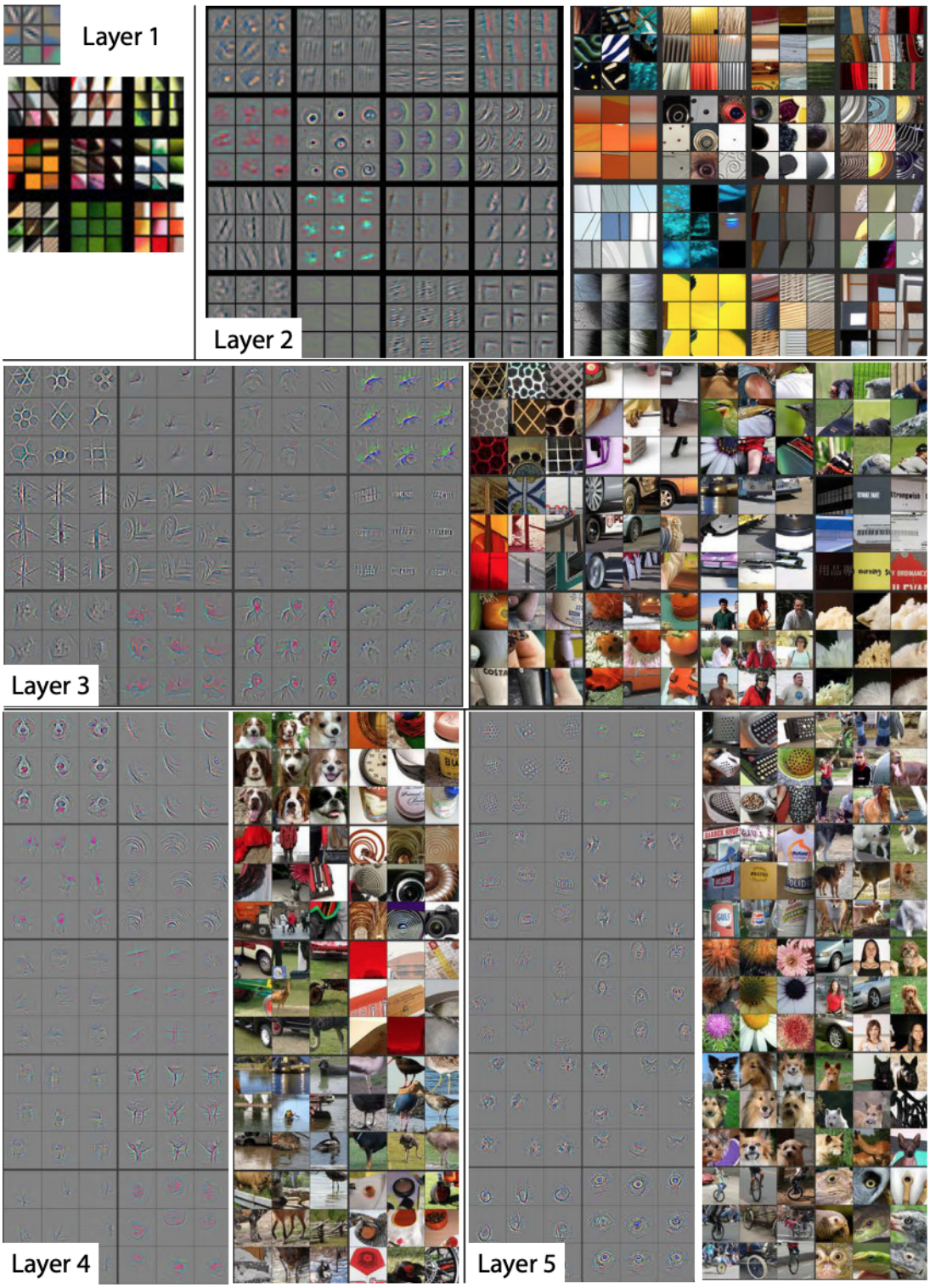

Figure 0.2: Visualization of features in a fully trained model. For layers 2-5 the top 9 activations in a random subset of feature maps are shown projected down to pixel space using the "deconvolutional" network introduced in **Zeiler2014** work

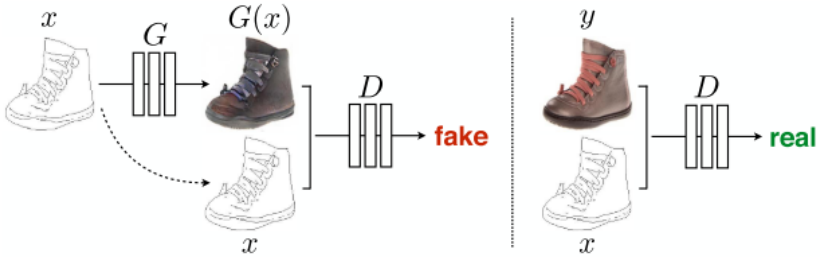

Figure 0.4: Training a conditional GAN to map edges to photo. The discriminator, D, learns to classify between fake and real edge, photo tuples. The generator, G, learns to fool the discriminator. Unlike an unconditional GAN, both the generator and discriminator observe the input edge map.

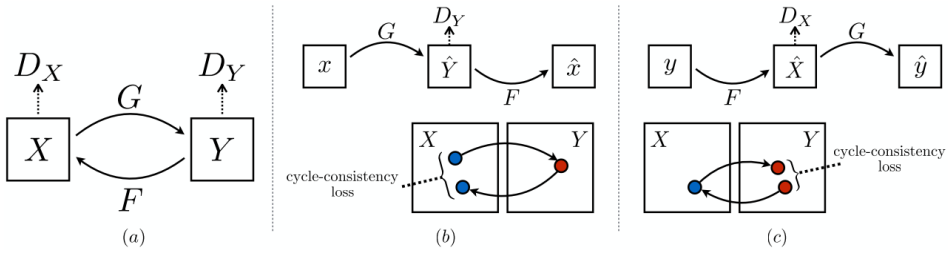

Figure 0.5: (Extracted from **Zhu2017a**) The mapping model denoted as $G_{X \to Y}$ in this work is is denoted in this figure as G and $G_{Y \to X}$ as F