GANs for Education: Automatic Generation of Learning Resources

João Tomás Caldeira
Supervisor: Prof. Cláudia Antunes

Abstract

Educators of data-driven courses, such as data science, make use of graphical representations of data (such as plots) as a means to convey certain intuitions about the subject. As Massive Open Online Courses (MOOCs) see a steep increase in use, the task of creating large amounts of educational content grows in relevance. We propose a deep learning approach whereby *generative adversarial networks* (GANs) are used to generate plots according to features specified by the user.

1 Introduction

Data science, statistics, machine learning, and related courses commonly make use of plots as a means to pictorially represent data. Scatter plots, box plots, and histograms are among the most common ways whereby we can extract information from a graphical representation of data. In particular, they may allow the viewer to intuitively get a sense of the distribution, correlation, etc. between variables that describe data. As such, they are an important resource for teaching courses which make use of data visualization in order to facilitate the communication of certain concepts, such as those mentioned above.

Massive Open Online Courses (MOOCs) have seen a recent rapid increase in utilization, both by universities and corporate entities [1]. They are a means to publish material related to a given subject (or course) being taught in an appropriate online space which can be accessed by the target audience (e.g., university students enrolled in a course). Materials may include lecture videos and notes, problem sets, assessments, etc.

MOOCs which address data-driven subjects, such as those mentioned in the beginning of this section, could thus benefit from tools with which to automatically generate graphical representations of data. In particular, generating a large amount of plots with specific characteristics would allow for students to be presented with virtually endless slightly different versions of the same problems. This has two obvious use cases: students could be presented with problems relevant to matters in which they are not proficient, and plots used in assessments could be made different for every student.

In this document, we propose the use of *generative adversarial networks* (GANs), conditioned on certain features of datasets found online, as a means to generate certain plots, e.g., scatter plots. This problem bears direct relevance to the task described above. Conditioning the models on data characteristics

(e.g., distribution and correlation between features) should allow a user to automatically generate plots that exhibit those characteristics to degrees which are parameterized by the user.

We will begin by conducting a short review of the relevant literature in section 2, including a brief historical overview of deep learning, the development of an understanding about a deep learning model called *neural networks*, and architectures which extend that model in a way that is relevant to this work. We will also highlight relevant work in the context of learning resource generation. The problem to address, as well as the proposed solution and the metrics with which to evaluate it, will be formalized in section 3.

2 Literature Review

According to Goodfellow et al. in their instant classic *Deep Learning* book [2], problems that are intellectually challenging for humans but straightforward for computers - those which can be formalized as a set of mathematical rules - were tackled and solved in the early days of artificial intelligence (AI). The problems which us humans solve intuitively - such as recognizing spoken words or faces in images - were those that proved to be the true challenge to AI. A solution to these problems involves allowing computers to learn a hierarchy of concepts, whereby they can learn complex concepts by building them out of simpler ones: "If we draw a graph showing how these concepts are built on top of each other, the graph is deep, with many layers. For this reason, we call this approach to AI deep learning".

This chapter will take a brief tour from primordial linear models of brain function, through the innards of a fully-fledged neural network, all the way to state-of-the-art architectures for generative models.

2.1 History of Deep Learning

Though many of the key concepts in deep learning are decades old, the usage of the term "deep learning" itself is rather recent. There are considered to have been three waves of development in the area: it was known as **cybernetics** in the 1940s-1960s, **connectionism** in the 1980s-1990s, and has been known as *deep learning* since the dawn of the third wave, in 2006.

2.1.1 Cybernetics

Indeed, the earliest mathematical model of a neuron was that of McCulloch and Pitts, created as long ago as 1943 [3]. The *McCulloch-Pitts neuron* was an early model of brain function, which attempted to provide an abstract formulation for the functioning of a neuron without necessarily attempting to simulate the real biological mechanisms of neurons [4].

It can be defined with a simple set of rules:

- It has a set of D excitatory binary inputs, such that $d \in \{0, 1\}, \forall d \in D$. In more mathematical terms, the input to the neuron is a d-dimensional vector of binary elements.
- It has a binary output y, where $y = 0$ is analogous to the neuron being at rest and $y = 1$ to it firing.

- If the sum of its D inputs is larger than a threshold value t, the neuron "fires" (i.e. $y = 1$). Otherwise, it stays at rest (i.e. $y = 0$).
- It has a binary inhibitory input i such that, if $i = 1$, the neuron doesn't fire (i.e., is inhibited). Conversely, if $i = 0$, the neuron may fire.

The functioning of the McCulloch-Pitts neuron is shown in figure 1:

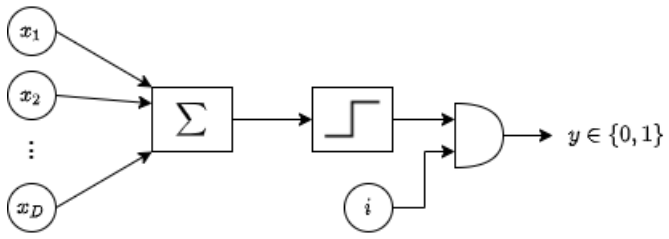

Figure 1: The McCulloch-Pitts Neuron

In a compact, formalized manner, the neuron is a function defined as:

$$f(\boldsymbol{x}) = y = \begin{cases} 1, & \text{if } \sum_{d=1}^{D} x_d > t \text{ and } i = 0, \\ 0, & \text{otherwise} \end{cases}$$

In 1949, Donald Hebb brought forth, in his book "The Organization of Behavior" [5], the theory of synaptic plasticity, whereby neurons adapt during the training process. This form of biologically motivated learning inspired psychologist Frank Rosenblatt to create the *perceptron* [6] [7].

Though similar to the McCulloch-Pitts neuron in concept, the perceptron introduced changes that proved to be major improvements, and which yet hold in modern neural networks:

- Inputs are now scaled by weights, whose value is a real number. This means that not only do inputs not necessarily contribute equally (i.e., being restricted to unity), but also that they can now have *inhibitory* effects on the perceptron.
- The neuron also takes a new constant (e.g. valued 1) input which, much like the original variable inputs, is also scaled by a weight. The resulting value is known as the *bias*.
- The inhibitory input is no longer present.

The perceptron is illustrated in figure 2.
In mathematical terms, the perceptron is a function f such that:

$$f(\boldsymbol{x}) = \begin{cases} 1, & \text{if } \boldsymbol{w}^\top \boldsymbol{x} + b \geq 0, \\ 0, & \text{otherwise} \end{cases}$$

This is the Heaviside function, but there are other possible formulations of the perceptron, such as the signum function [4]. In whichever case, the decision boundary of the perceptron is a hyperplane:

$$b + w_1 x_1 + w_2 x_2 + ... + w_D x_D = 0 \Leftrightarrow b + \boldsymbol{w}^\top \boldsymbol{x} = 0$$

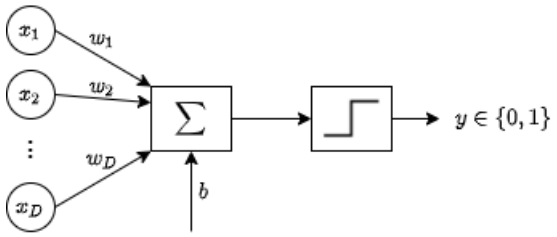

Figure 2: Rosenblatt's perceptron

This means that, in the context of binary classification, the perceptron can only achieve perfect accuracy in tasks where classes are linearly separable, i.e., where the separatrix is a straight line.

Perhaps more importantly than the aforementioned changes, Rosenblatt presented an algorithm whereby the perceptron could learn the weights $\boldsymbol{w}$ from examples (algorithm 1).

Algorithm 1: Perceptron Learning Algorithm

Input: $\{\boldsymbol{x}_n, y_n\}_{n=1}^{N}$
// b and $\boldsymbol{w}$ are initialized randomly
while *not converged* **do**
  for n = *1, N* **do**
    $error = y_n - f(\boldsymbol{x}_n)$
    if $error \neq 0$ **then**
      $\boldsymbol{w} = \boldsymbol{w} + error \times \boldsymbol{x}_n$
      $b = b + error$
    end
  end
end

In 1960, Widrow and Hoff presented the *adaptive linear element* - or ADALINE. It was quite similar to the perceptron, with the chief difference being in the learning rule:

Algorithm 2: ADALINE Learning Algorithm

Input: $\{\boldsymbol{x}_n, y_n\}_{n=1}^{N}$
// b and $\boldsymbol{w}$ are initialized randomly
while *not converged* **do**
  for n = *1, N* **do**
    $error = y_n - (\boldsymbol{w}^\top \boldsymbol{x}_n + b)$
    if $error \neq 0$ **then**
      $\boldsymbol{w} = \boldsymbol{w} + error \times \boldsymbol{x}_n$
      $b = b + error$
    end
  end
end

While the perceptron uses its binary output ($f(\boldsymbol{x}) \in \{0, 1\}$) to compute the error used to update the weights, ADALINE uses the continuous net value before applying the activation function (e.g., Heaviside, signum, etc.). As such, ADA-

LINE's updates are more representative of the actual prediction error. ADALINE's learning rule is a special case of *stochastic gradient descent*, which remains the backbone of many of the most popular modern deep learning models.

Gradient descent is a first-order, iterative optimization algorithm, used to find a set of parameters (or *weights*, as known in deep learning literature) which minimize a *cost* (or *loss*) function. We call it a first-order optimization algorithm since it uses only first-order derivatives as opposed to, for example, Newton's method, which uses the Hessian of a function. It is an iterative method since it sequentially updates function parameters to achieve improved approximations. Let f be the function that we are using as a model (e.g., a linear function $f(\boldsymbol{x}) = w_0 + w_1x_1 + ... + w_Dx_D$ used for regression) and $\boldsymbol{w}$ its set of weights. Given an arbitrary loss function $\mathcal{L}$ (e.g., the mean squared error), an input x and its *target* value y, the gradient descent update to $\boldsymbol{w}$ can be written as:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \gamma \nabla_{\boldsymbol{w}_t} \mathcal{L}(f(x), y), \tag{1}$$

where γ is the *learning rate* - some scalar usually between 0 and 1, which scales how weights change in response to the computed error. We can say, in the context of machine learning, that the loss function $\mathcal{L}$ estimates how well the current model's predictions match the distribution of the target variables in the observed data. Recall that the gradient of a function gives us the "direction of steepest ascent". By *subtracting* the gradient of $\mathcal{L}$ with respect to $\boldsymbol{w}$ to the current weights $\boldsymbol{w}_n$, we are changing the values of $\boldsymbol{w}$ so that the loss function $\mathcal{L}$ is *minimized*. The above rule uses the negative of the direction of steepest ascent to update the weights, hence gradient *descent*.

When using the term "gradient descent" it is usually implied that, in each update, we calculate the loss function over the entire dataset. For large datasets, this can be computationally expensive.

Stochastic gradient descent (SGD) takes a subset of the data and computes the loss over that subset, instead of the whole dataset. The term "stochastic" comes from the fact that the loss over a subset of the dataset is a stochastic approximation of the loss over the dataset. Some authors differentiate between gradient descent methods whose update uses only one sample (simply SGD) from methods using a subset of two or more samples, the latter being known as *minibatch gradient descent*.

In 1969, Minsky and Papert [8] exposed some of the limitations of linear models, which the perceptron and ADALINE were. Famously, linear models cannot learn the XOR function, since it is not linearly separable. This caused a backlash against biologically-inspired learning, which put the development and research of neural networks mostly on hold. This "pause" is known as the first "AI winter".

2.1.2 Connectionism

The second wave of neural network research lasted from the 1980s up to the mid 1990s. It was called *parallel distributed processing* or, more famously, *connectionism*. The central idea in connectionism is that a large number of interconnected, simple computational units can achieve emergent intelligence [9] [10].

In 1980, Fukushima introduced the *Neocognitron* [11], a powerful deep neural network for image pattern recognition. It was arguably the second deep neural

network, after Ivakhnenko's GMDH networks [12]. The impact of the Neocognitron was twofold: it brought more interest to the development and research of deep learning and became the basis for what we call the *convolutional neural network* (CNN) [13], with some arguing it to be the first of that kind [4].

A major breakthrough during the connectionist movement was using *back-propagation* to train deep neural networks, which successfully generated useful internal representations [14] [15]. The back-propagation algorithm has since been the dominant method of training neural networks.

In 1991, Hochreiter [16] formally showed that deep neural networks were hard to train with back-propagated errors: those either shrink rapidly or grow rapidly along layers. Those problems are now known as *vanishing gradients* and *exploding gradients*, respectively. Though there are a number of remedies for these problems, they are still an area of active research and enumerating them is outside the scope of this text.

The problems above were among the difficulties in modeling long sequences as identified by Bengio et al., in 1994 [17]. Some of those difficulties were addressed by the *Long Short-Term Memory*, a recurrent neural network model created by Hochreiter and Schmidhuber in 1997 [18].

This second wave of research in neural networks lasted until the mid-1990s [4]. At the time, ventures based on neural networks began making unrealistically ambitious claims, which research failed to fulfill [2]. With other machine learning methods (notably kernel machines) achieving good results on many important tasks [19] [20], the popularity of neural networks dipped until 2006.

2.1.3 Contemporary Deep Learning

Hinton et al. showed, in 2006, that a strategy called *greedy layer-wise pre-training* could be used to train *deep belief networks*, which are a type of neural networks, efficiently [21]. It was quickly shown that the same method could be used to train other types of deep networks [22] [23]. This was the breakthrough that started the third wave of neural networks research, now under the name of *deep learning*, which emphasized the theoretical importance of network depth and how it was now possible to train deeper neural networks. This wave of neural network development is ongoing at the time of writing.

2.2 Multilayer Perceptrons

Multilayer perceptrons (MLPs), also commonly called *feed-forward neural networks*, are the quintessential example of a modern deep learning model. In this section, we explain how MLPs work.

2.2.1 Modern Artificial Neuron

The original perceptron only had a single layer, i.e., had none of what we now call *hidden layers*. We usually do not call neural networks *deep* unless they have at least one hidden layer. Each layer is a set of neurons. The structure of a modern neuron is as depicted in figure 3.

We can see the similarities between a modern artificial neuron and the original perceptron: the neuron receives D weighed inputs, has a bias, and sums all those terms into what is depicted in figure 3 as *net*. Finally, some *activation*

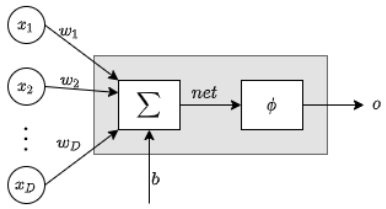

Figure 3: Modern artificial neuron

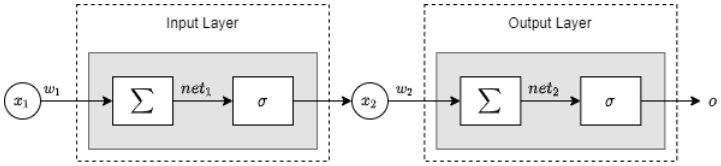

Figure 4: The world's simplest net

function ϕ is applied to *net* and the result is the output of the neuron. Formally, a modern artificial neuron is just a function

$$Neuron(\boldsymbol{x}) = \phi(\boldsymbol{w}^\top \boldsymbol{x} + b), \tag{2}$$

where ϕ is usually a nonlinear function, for reasons we will see below.

2.2.2 The World's Simplest Net

Hopefully it has now been made clear how a single neuron works. To understand all the mechanisms at work in even the largest neural networks we only need to observe a two-layer network (i.e., no hidden layers), with a single neuron in each layer. Let us consider the world's simplest net[1] in figure 4.

There are a few new things introduced in this figure:

- The input to the first neuron is 1-dimensional, i.e., a scalar. The summation operator is not really doing anything in this case, since its input is just a single element $x_1 \times w_1$. As such, the input to the activation function is $net_1 = x_1 \times w_1$.
- Speaking of the activation function; while ϕ (greek letter *phi*) is used to refer to any activation function, we arbitrarily chose the sigmoid (represented by the greek letter *sigma*) function as an activation for both layers.
- We eliminated the bias term b. The bias term can be replaced by an extra entry w_0 on the weight vector and a corresponding 1 appended to the input vector.
- The output of the first neuron is not the output of the network. Instead, it is the input to the second neuron, which in turn will produce the output of the network o.

[1] Inspired by 'Lecture 15. Learning from examples - Neural approaches', taught by F. Melo at Instituto Superior Técnico, *Planning, Learning, and Intelligent Decision-Making*, 11/2020

Considering the case where there are no activation functions in the neural network shown in figure 4, the output of the net for a given input x would be:

$$SimplestNet(x) = w_2 w_1 x, \tag{3}$$

which we can generalize for any i-th layer of a network:

$$Neuron(x)_i = w_i Neuron(x)_{i-1}. \tag{4}$$

We can see how, in this case, the output of a network would be the result of a composition of linear functions. If the bias term is considered, then it would be a composition of affine functions, which are just linear transformations with nonzero translation - though distinguishing between the two is not relevant in this context.

We know that a composition of linear functions is equivalent to a single linear transformation. Linear functions have a number of limitations (cf. the XOR problem), as they can't establish nonlinear decision boundaries. Using a nonlinear activation function, such as the sigmoid, allows us to approximate a larger number of functions. Indeed, any continuous function on the unit cube, i.e., whose domain is $[0, 1]^n$, can be approximated by a one-layered neural network with a sigmoidal activation function [24].

The sigmoid function and its derivative are, respectively:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

$$\sigma'(x) = \sigma(x)\dot{(}1 - \sigma(x)) \tag{6}$$

As such, this neural network can be defined as a composition of functions of the neuron on the left, N_1, and the neuron on the right, N_2:

$$SimplestNet(x_1) = (N_2 \circ N_1)(x_1) = \sigma(w_2 \sigma(w_1 x_1)) \tag{7}$$

The function above is known as a *forward pass*, which is really just feeding some input to the network and getting some output. However, we need to train the network so that it learns "good" weights. We have seen that loss functions estimate how well the current model's predictions match the distribution of the target variables in the observed data. In the context of neural networks, "good" weights are those which, when parameterizing a model, yield "good" predictions by some arbitrary measure.

Neural networks are one of many models which are optimized under the *maximum likelihood estimation* probabilistic framework, which involves finding a set of parameters (weights, in the context of neural networks) which are good at explaining observed data. To do that, we select some *likelihood function*, which quantifies the probability of a set of observations to occur given the parameters of some model. One such function is the *negative log-likelihood*, also known as simply *log loss*:

$$\mathcal{L}(o, y) = -ylog(o) - (1 - y)log(1 - o) \tag{8}$$

The loss function above is a *cross-entropy* between the model's currently "perceived" distribution and the empirical distribution defined by the training set. As such, the term "cross-entropy" is commonly used interchangeably with negative log-likelihood and log loss.

2.2.3 Backpropagation

Having defined the activation for both of the neurons in our net (figure 4) as well as a loss function, we can derive a learning rule for the network. The backbone of the most widely used optimization algorithms in neural networks is gradient descent, whereby we update the set of weights according to a gradient of the chosen loss function. Indeed, to update w_1 and w_2, we need the gradient of the negative log-likelihood with respect to those weights:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial o} \times \frac{\partial o}{\partial w_1} \tag{9}$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial o} \times \frac{\partial o}{\partial w_2} \tag{10}$$

Using the chain rule of derivatives and simplifying, we get:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \underbrace{(o-y)w_2 x_2(1-x_2)}_{\delta_1} x_1 \tag{11}$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = \underbrace{(o-y)}_{\delta_2} x_2 \tag{12}$$

We can see how the derivative of the loss function with respect to w_i is always "something" scaled by x_i. We refer to that "something" as δ_i, as depicted above. Observing that δ_1 depends on δ_2, and recalling the derivative of the sigmoid in equation 6:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \underbrace{\overbrace{(o-y)}^{\delta_2} w_2 \overbrace{x_2(1-x_2)}^{\sigma'(x_2)}}_{\delta_1} x_1 \tag{13}$$

We have seen how a single-layer network can learn weights using gradient descent. Indeed, the output layer can use the gradient descent update rule to learn its weights as shown in equation 1, where

$$\nabla_{\boldsymbol{w}} \mathcal{L}(f(x), y) = (f(x) - y)x. \tag{14}$$

However, for layers anterior to the output layer, the gradient descent update is always going to depend on the gradient of the previous layer. Using the terms in equation 13, we can write the expression for the gradient of the i-th layer (except the output layer) with respect to the gradient of the posterior layer:

$$\frac{\partial \mathcal{L}}{\partial w_i} = \delta_i x_i, \qquad \text{where} \qquad \delta_i = \delta_{i+1} w_{i+1} \sigma'(x_{i+1}). \tag{15}$$

2.2.4 Activation Functions

The simplest activation function is the identity. We may also refer to this as a *linear activation* or even that there is no activation at all. In layers with linear activations, their output is just

$$Linear(x) = w^\top x + b, \tag{16}$$

where x is the input to the layer. This is unlike all the other examples in this section, where we will assume their input is already the dot product (plus bias) between the input to the layer and its weights. Linear functions are most commonly seen in output layers where the network is addressing a regression task.

The *softmax* function is used almost exclusively in the output layer of a function. It outputs a vector with the predicted probabilities for each class,

with the constraint that the probabilities need to sum to 1. Given some vector x with the unconstrained and denormalized predicted probabilities, the softmax computes

$$Softmax(x) = \frac{e^x}{\sum_i e^{x_i}}. \tag{17}$$

Softmax is differentiable. It always assigns some probability to all elements of the prediction vector, even if very small. Furthermore, because of the natural exponentiation, the probabilities of the biggest scores are increased and the probabilities of the lowest scores are decreased when compared to standard normalization. These properties make softmax a very suitable function for tasks where we want to predict which (single) class an object belongs to, which is known as *multiclass classification*.

We have used a sigmoid as a nonlinearity in our example network. Sigmoids were historically popular as they were thought to behave similarly to a neuron with a saturating "firing rate" - which was later found to be a poor interpretation, as we will see below. They also constrained their output to $]0, 1[$, which was interesting from a probabilistic point of view. However, there are some issues with sigmoids.

First of all, the exponentiation operation is a bit computationally expensive. Perhaps more importantly, its output is not zero-centered - in fact, it is always positive. This means that the input for a layer that follows a sigmoid nonlinearity will be positive. In other words, if we have

$$net = \boldsymbol{w}^\top \boldsymbol{x} + b, \tag{18}$$

then $\boldsymbol{x} > 0$ element-wise. If we are updating the weights of a layer that follows a sigmoid nonlinearity, we will have

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial net}\frac{\partial net}{\partial w} = \frac{\partial \mathcal{L}}{\partial net}\boldsymbol{x}, \text{ with } \boldsymbol{x} > 0. \tag{19}$$

Since $\frac{\partial \mathcal{L}}{\partial net}$ is a scalar, then the gradient update to that layer's weights $\frac{\partial \mathcal{L}}{\partial w}$ will be either all positive or all negative, leading to "zig-zagging" updates as we may visualize in figure 5.

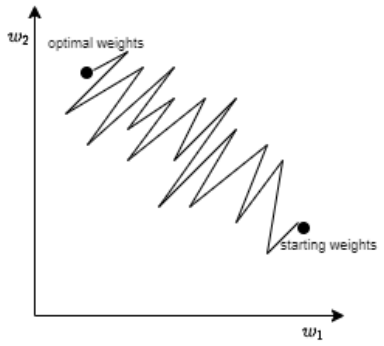

Figure 5: Zig-zagging towards the target weights

In the example above, we see how weight updates being either all-negative or all-positive only allows for movements northeastwards or southwestwards. This issue is mitigated in the case where gradients are obtained for a batch of samples, whereby the sum of the various individual gradients can introduce variable signs in the final gradient.

Most famously, sigmoid units often saturate and "kill" gradients. It is an undesirable property of the sigmoid that, when its input is either a large positive or large negative value, the activation becomes very close to either 0 or 1. This can be observed in figure 6.

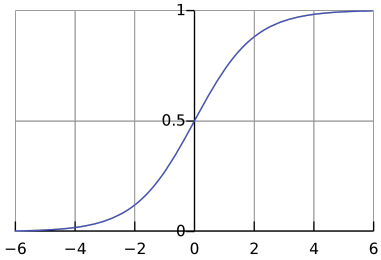

Figure 6: Plot of the sigmoid function

The plot above shows how values quickly become very close to 0 or 1 if they are somewhat negative or positive. Recall equation 15, whereby we see that the update to a layer's weights depends on its own output via the derivative of its activation function. Recall also the derivative of the sigmoid (equation 6), and it is now trivial to see why numbers at either "tail" of the sigmoid would saturate the gradient:

$$\sigma'(5) = \sigma(5)(1 - \sigma(5)) \approx 0.993(1 - 0.993) \approx 0.007 \tag{20}$$

For a relatively small input such as 5, the sigmoid's derivative for that input is very close to zero. Since this derivative scales the gradient used to perform weight updates, we can see how a sigmoid layer could rapidly shrink the error signal propagated towards previous layers, thereby effectively "killing" the gradient.

This gradient shrinkage throughout backpropagation is known as the *vanishing gradient* problem. Though there are multiple factors which contribute to how gradients flow through the network, sigmoids have mostly fallen out of use in hidden layers due to the above issue. Instead, since we can view them as yielding a probability, they are used to make binary decisions, e.g., in a binary classification scenario, or even for making several "yes or no" predictions regarding multiple targets in the same input. The latter task is known as *multilabel classification*.

The *tanh* function (hyperbolic tangent), whose equation we can see below, performs better than the sigmoid in practice.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{21}$$

Relatively to sigmoid units, *tanh* units have zero-centered output. However, they can be seen and written as just a scaled sigmoid

$$tanh(x) = 2\sigma(2x) - 1, \tag{22}$$

and thus also saturate and kill gradients.

The *rectified linear unit* (ReLU) was used to great success in AlexNet [25], a historically important CNN, and would remain a standard choice for activation units for NN architectures of all kinds. It computes

$$ReLU(x) = max(0, x), \tag{23}$$

which is very efficient, and also does not saturate for positive inputs. It is also a closer approximation to the real activation of a neuron than the sigmoid, in the sense that they remain fully dormant until there is enough stimulus to fire. However, unlike real neurons, which are "all-or-nothing", ReLU units have variable output within their positive region.

ReLUs do not have zero-centered output, which we have seen is undesirable. Furthermore, they can still saturate and kill gradients if their inputs fall on the negative region.

The *Leaky ReLU* [26] differs from the ReLU only in that it does not "die". If the input is negative, it is scaled by some small factor:

$$LeakyReLU(x) = max(0.01x, x), \tag{24}$$

and this slight negative slope prevents Leaky ReLU units from saturating. Due to the slope on the negative region, its outputs also get closer to having a zero mean.

The *parametric ReLU* (PReLU) [27] is very similar to the Leaky ReLU, except that the factor by which it scales the negative region of the input is a learned parameter itself:

$$PReLU(x) = max(\alpha x, x), \tag{25}$$

where α is also trained via backpropagation.

The *exponential linear unit* (ELU) [28] carries all the benefits of ReLU, but handles the negative region differently:

$$ELU(x) = max(\alpha(e^x - 1), x). \tag{26}$$

When compared to the Leaky ReLU, it saturates on the negative region, which is argued to add robustness to noise. A disadvantage of ELU is that it introduces an exponentiation, which we have seen to be computationally expensive. It is noteworthy that, contrarily to PReLU units, the α parameter is not learned in ELUs - instead, it is just another hyperparameter.

Finally, the *maxout* neuron generalizes ReLU and Leaky ReLU:

$$Maxout(x) = max(w_1^\top x + b_1, w_2^\top x + b_2), \tag{27}$$

where w_i and b_i are network weights and biases, respectively. Maxout neurons do not saturate nor die, but incur the doubling of the number of parameters per neuron.

2.2.5 Regularization

Occam's razor, also known as the *principle of parsimony*, states that "entities should not be multiplied beyond necessity". It is, in the context of scientific methodology, commonly paraphrased as "among competing hypotheses, the simplest is the best".

A nice interpretation of Occam's razor in the context of machine learning is that if we have several models that perform similarly in explaining current observations, then the simplest one is the best: since it makes the fewest assumptions, it is more likely to generalize its explanatory power to unseen observations. *Regularization* is how we penalize models for making too many assumptions in order to explain current observations.

A common way to regularize a neural network is to add an explicit term to the loss function that penalizes large weights:

$$\mathcal{L}(w) = \frac{1}{M}\sum_{i=1}^{M} \underbrace{\mathcal{L}(f(x_i, w), y_i)}_{Data\ loss} + \underbrace{\lambda R(w)}_{Reg.\ term} \ . \tag{28}$$

The data loss tells the model how its predictions ought to match the training data. For example, we have used the cross-entropy loss in the "world's simplest net" above. The regularization term, on the other hand, introduces some penalty over the weights which "tells" the model to be simple so that it can work well on test (unseen) data. The λ is a hyperparameter of the network. The two most common formulations for the regularizing term are the *L1 loss* and the *L2 loss*:

$$R_{L1}(w) = \underbrace{\|w\|_1}_{1-norm} \tag{29}$$

$$R_{L2}(w) = \underbrace{\|w\|_2^2}_{2-norm} \tag{30}$$

L1 regularization tends to lead to sparser solutions, i.e., enforces a greater amount of zero weights [29]. L2 regularization, also known as *Tikhonov regularization* and *weight decay*, on the other hand, yields diminishing penalties as weights move closer to zero, thereby discouraging sparsity.

Another form of regularization that is commonly seen in neural networks is *dropout* [30]. Dropout consists of "dropping" neurons with a chosen probability during a forward pass. To "drop" a neuron during a forward pass consists of setting its activation for that pass to zero. The probability with which each neuron is dropped during a forward pass is a hyperparameter to the network.

Dropout has a nice interpretation where we can say that it forces the network to have redundant representations, thereby preventing co-adaptation of features[2]. We can use as an example a network that is trying to classify dogs. When making a prediction about whether a sample is a dog, the model cannot depend too much on a single feature. Instead, it needs to distribute what it perceives as "dogness" across many different features. For example, if the model relies on the presence of both a dog-ish tail and dog-ish ears to make a prediction, then a sample where the tail isn't visible will increase the probability that the model makes a mistake.

By randomly dropping neurons at training time with a probability p, we introduce stochasticity into the network. However, that stochasticity may be

[2]Inspired by 'Lecture 7. Training Neural Networks II', taught by J. Johnson at Stanford University School of Engineering, *CS231n*, 08/2017

undesired at testing or deployment time. In practice, a cheap way to eliminate stochasticity at test time is to scale the activations by p. However, that introduces a small overhead in that activations need undergo an extra multiplication. The best solution is thus probably "inverted dropout", whereby instead of multiplying activations by p on test time, we divide activations by p at training time.

NNs, especially deep ones, are very sensitive to weight initialization and hyperparameters of the learning algorithm. Furthermore, the distribution of each layer's inputs changes during training as the parameters of the previous layer change. This phenomenon is known as *internal covariate shift*. These issues were addressed in a 2015 paper that introduced *batch normalization* [31].

In batch normalization, the empirical mean and variance for each dimension are computed for each minibatch in a forward pass. This is usually done after each fully-connected layer (i.e., applied to $w^\top x + b$) and before a nonlinearity is applied. Batch normalization for a batch of samples $\hat{X}$ is

$$BatchNorm(\hat{X}) = \frac{\hat{X} - \mathbb{E}[\hat{X}]}{\sqrt{Var[\hat{X}]}}. \tag{31}$$

The above function is differentiable, so it can be backpropagated through. Furthermore, it is common in practice to allow the layer to learn two parameters which scale and shift the normalized batch $\hat{X}$:

$$FinalBatchNorm(\hat{X}) = \gamma\hat{X} + \beta. \tag{32}$$

The γ is not to be confused with the learning rate in gradient descent. While during training the empirical mean and variance are computed for each batch, at test time the batches are normalized using mean and variance computed over the entire training data.

Batch normalization improves gradient flow through the network, reduces dependence on weight initialization, allows for greater learning rates, and has been observed to improve generalization ability [32]. Some authors argue that the improvements introduced by batch normalization are due to a smoothening of the optimization problem landscape [33].

2.2.6 Modern Practices in Neural Networks

The concepts discussed thus far are extensible to MLPs with any amount of nodes and hidden layers. However, there are a few noteworthy modern practices:

- Though SGD itself is a very popular optimization algorithm, it is very common to find other SGD-based methods employed in neural networks such as Momentum [34], Adagrad [35], Adam [36], etc.
- Hidden layers usually do not employ sigmoid functions (except for *recurrent neural networks*), but rather a Rectified Linear Unit (ReLU) [37] or its undying cousin Leaky ReLU [26].
- For output layers, the chosen activation functions are usually as follows, depending on the problem at hand:
  - Regression: single output node, linear activation.

- Binary classification: single output node, sigmoid activation.
- Multilabel classification: one node per label, sigmoid activation.
- Multiclass classification: one node per class, softmax activation.

A review of modern practices is beyond the scope of this document, and as such we will only pay closer attention to methods employed in the neural network architectures discussed in the next chapters.

2.3 Convolutional Neural Networks

The MLP is the archetypal example of a neural network architecture, as we have seen in the small historic overview of DL above. However, one of the first architectures that could be considered deep was the Neocognitron [11], which drew from neurophysiological insights [38]: it was found that cats have simple and complex cells in their visual cortex, which respond to properties in their sensory input [39] [40]. Complex cells were less sensitive to the position of objects in their receptive field, i.e., they were less spatially invariant. This draws a nice analogy to the way we like to think CNNs work - that the first layers respond to low-level features (such as lines and edges), and that deeper layers are able to learn more abstract features (such as shapes and objects).

CNNs perform well under the assumption that points that are close to each other in the data share an interesting relationship. As such, they have been used in speech recognition [41], text classification [42], and time series forecasting [43], among other applications.

For the purpose of demonstrating the basic functioning of a CNN, we will assume that the inputs to the model are images. Images may be 2- ($width \times height$) or 3- ($width \times height \times depth$) dimensional. The depth of an image is the amount of color channels that describe it. For example, a grayscale image may have a single color channel, thus being 2-dimensional. An RGB image, on the other hand, may have 3 color channels (red, green, and blue), making it a 3-dimensional image with a depth of 3.

2.3.1 Convolutional Layers

Chief among the differences between MLPs and CNNs is the use of *convolutional layers*. A convolutional layer aims to preserve the input's spatial structure by convolving a *filter* with it.

A filter (or *kernel*) is a set of weights with the same dimensionality as their receptive fields. The *receptive field* of a filter is a region of the input which can influence it. Filters are typically spatially small relatively to the image and take the shape of a square (i.e., $width = height$). However, they span the entirety of the depth of the image. For example, given an RGB image with shape $32 \times 32 \times 3$, a reasonable filter shape for the first layer of a CNN would take the shape $5 \times 5 \times 3$.

The convolution operation itself can be described as "sliding" filters spatially over the image, computing the dot products between the filter's weights and the image's pixel values, as pictured in the 2-dimensional example in figure 7.

The filter starts at the top left of the image, computes the dot product between its weights and the shown chunk of the image, and shifts a pixel to the right. When the filter reaches the rightmost border of the image, it shifts

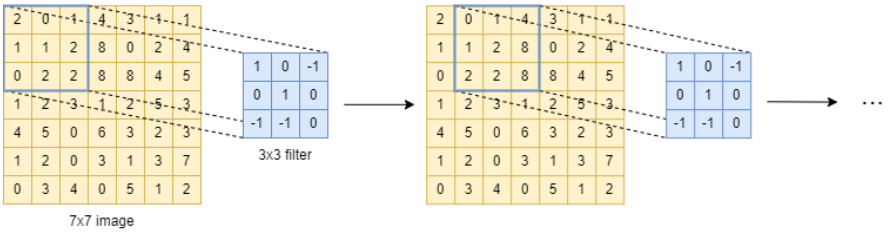

Figure 7: The convolution operation

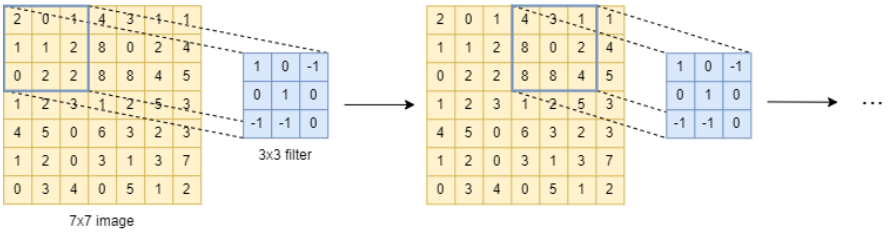

Figure 8: Convoluting with a stride of 3

a pixel downwards and returns to the left of the image. In this case, the filter would shift rightwards 4 times (i.e., perform 5 dot products in the first row) before shifting a pixel downwards. In total, it would need to shift downwards 4 times so that it reaches the end (bottom right) of the image.

The dot product between a filter and a chunk of an image with the same size yields a scalar. If we store those scalars in a matrix whose cells' positions correspond to the position of a filter relative to an image when performing a convolution, we get an *activation map*. In the example depicted above, the resulting activation map would have a shape of 5×5, since the filter has space to shift 4 pixels horizontally and 4 pixels vertically.

Keep in mind that, for each filter that is convoluted over the input, an activation map is generated. As such, the depth of the output of a convolutional layer is equal to the number of activation maps that are generated. One may imagine that the output of a convolutional layer consists of all the activation maps stacked on top of each other.

In the example above, we shift the filter pixel by pixel. It may be, however, that we want to slide the filter by more than a single pixel at a time. The number of pixels by which we slide the filter through the image is called the *stride* of the convolution, and it is a hyperparameter of the convolutional layer. Using the same image and filter as above, but convoluting with 3-pixel shifts at a time (i.e., stride 3), we would have a filter convoluting over the input as shown in figure 8.

In this case, after computing the dot product between the first chunk of the image and the filter, the latter shifts to the right with a stride of 3 pixels.

It may be that, due to the size of the image and the chosen filter size and stride value, the filter doesn't "fit" into the input symmetrically. In the image above, if we imagine trying to slide the filter once more with a stride of 3, we can see how its receptive field would be placed partially outside the input.

To address this issue, a common approach is to "pad" the input with some constant so that the filter fits neatly into the input. Particularly, we almost always use zeroes to pad, which is why this operation is typically referred to as *zero-padding*, whose size is a hyperparameter of the convolutional layer. We can formulate the size of an activation map A with respect to the size of the input X, the stride value S, and the zero-padding size P:

$$A = \frac{X - F + 2P}{S} + 1. \tag{33}$$

So, for the convolution operation in figure 8, we have:

$$A = \frac{7 - 3 + 2 \times 0}{3} + 1 = 2.\overline{3}. \tag{34}$$

Since the result is not an integer, it indicates that the filter cannot be applied symmetrically across the input. However, if we add a zero-padding of 1:

$$A = \frac{7 - 3 + 2 \times 1}{3} + 1 = 3. \tag{35}$$

Viewing the usage of zero-padding pictorially:

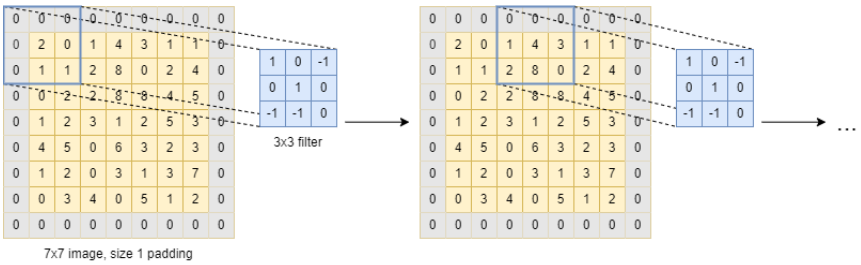

Figure 9: Convoluting with a stride of 3 and zero-padding

The usefulness of padding is twofold: on the one hand, as we have seen above, it helps preserve features that lie at the edges of the input by allowing filters to stride along its entire width and height. On the other hand, it also allows for greater control over the spatial dimensions of the layer's output by increasing the input's width and height.

We have now discussed all the hyperparameters of a convolutional layer:

- Filter size: the length, in pixels, of a spatial edge of the filter.
- The number of filters to slide over the input. Corresponds to the amount of activation maps of that layer, and thus to the depth of its output.
- Stride: the amount, in pixels, by which a filter shifts over the input. We can think of the stride as a ratio between how much we move in the input and how much we move in the output.
- Padding: the amount, in pixels, to pad each spatial edge of the input.

It is very common to apply some nonlinear transformation to the output of a convolutional layer; most commonly a ReLU or one of its cousins Leaky

ReLU [26], PReLU [27], etc. In other words, convolutional layers will usually appear followed by a layer which introduces some nonlinear transformation, e.g., by applying ReLU element-wise to the output of the convolutional layer.

Recall that in each layer of an MLP, each neuron is connected to all the neurons in the previous layer. For that reason, we refer to those layers as *fully-connected layers*. If a layer has n neurons and the next has m neurons, the connections between those layers will be parametrized by $(n+1) \times m$ weights.

Yann LeCun pioneered the first successful applications of CNNs in the 1990's, of which LeNet [13] was the best known. However, it was AlexNet [25], taking first place in the ImageNet ILSVRC challenge of 2012 and beating the runner-up by around 10%, which popularized the use of CNNs in computer vision. Let us use AlexNet as a case study to observe an important aspect of convolutional layers: *weight sharing*.

AlexNet accepted images of size $227 \times 227 \times 3$. The neurons in its first layer had a filter size of 11, a stride of 4, and zero padding. Using equation 33, we can calculate the size of the activation maps of the first layer: $A = \frac{227-11}{4} + 1 = 55$. It also had a depth of 96; so, for 96 activation maps, there would be $55 \times 55 \times 96 = 290,400$ neurons in the first layer. With each neuron being connected to an $11 \times 11 \times 3$ receptive field, the first layer would be parametrized by $290,400 \times 11 \times 11 \times 3 = 105,415,200$ weights, excluding biases. Computations in networks of this scale would be intractable.

Central to the functioning of CNNs is the assumption that, if some feature of interest is found in a certain position of the input, then it will also be of interest if found somewhere else on the input. In practice, this means that neurons in the same "depth slice" of a given convolutional layer share their weights. Keeping to the example of AlexNet, we have seen that the volume of the first layer would be $55 \times 55 \times 96$. If we constrain all the neurons in the same depth slice to share the same set of weights, then there are only 96 sets of weights. Since the receptive field of each neuron is an $11 \times 11 \times 3$ region of the input, the first layer would have $96 \times 11 \times 11 \times 3 = 34,848$ weights. If we also account for biases, then the first layer has a total of $34,848 + 96 = 34,944$ parameters. This concept of weight sharing allows for reducing network complexity by orders of magnitude and is still central to current CNN architectures at the time of writing.

To summarize, a convolutional layer receives some input with dimensions ($width \times height \times depth$) and convolutes a set of filters spatially over it. The spatial dimensions of the output (i.e., the width and height) will depend on the values of some of the layer's hyperparameters: filter size, stride length, and padding size. Similarly, the depth of the output of the layer will be equal to the number of filters chosen for that layer. It is common to "stack" a ReLU layer on top of each convolutional layer.

2.3.2 Downsampling Layers

In the context of CNNs, *downsampling* is an operation that reduces input dimensionality. Two of the effects of downsampling are of particular interest:

- By reducing dimensionality, the number of parameters (weights) decreases, resulting in a reduction of computational complexity.

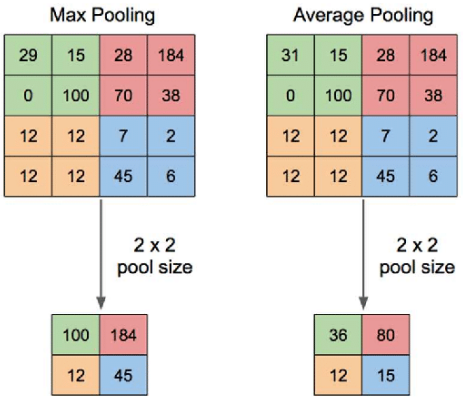

Figure 10: Max and average-pooling with size 2 and stride 2 [45]

- Spatial downsampling (i.e., dimensionality reduction in the width and height of the input) introduces spatial invariance.

It is noteworthy that the aforementioned spatial invariance comes mostly in the form of imperviousness to translation, and small tilts and distortions of the input. Elsewise, CNNs mostly remain sensitive to rotation [44]; which is commonly addressed with training data augmentation (e.g., introducing rotated versions of images).

The convolution operation itself can perform downsampling. Usually, convolutional layers shrink or maintain the spatial dimensions, but increase the number of filters per layer as the network gets deeper. The chosen values for the hyperparameters of the convolutional layer will determine its output shape. In particular, one may resort to using padding as a means to preserve spatial dimensionality.

Pooling is a spatial downsampling operation which we can view intuitively as trying to "summarize" the information in a receptive field using a single scalar. Much like in a convolution, pooling has a filter of fixed size slide over the input and produce a single scalar for each position it takes. However, in contrast with a convolutional filter, pooling is done with a depth of 1. This makes it so that the depth of the input is preserved, which is why we say pooling performs *spatial* downsampling specifically.

Many functions have been tried as a way to "summarize" the data in the pooling filter's receptive field. *Max-pooling* simply outputs the largest value in its filter, and *average-pooling* outputs the average of those values (figure 10).

Though there are more functions which would be viable for use in pooling, max-pooling in particular has been empirically shown to be the superior choice in models dealing with images due to their efficiency in capturing invariances [46].

Most pooling operations don't have any parameters which need to be trained, though there are some exceptions [47]. Rather, one only needs to specify the filter and padding size, as well as the stride value for the pooling layer.

We have seen how usually, deeper layers in a CNN have smaller spatial dimensions and greater depth (i.e., many activation maps). A method which allows for depth downsampling is the use of 1×1 convolutions, first investigated in 2013 [48]. By using a convolutional layer with a filter size of 1, stride 1, and no

padding, it is possible to reduce the depth of the previous layer's output by an arbitrary amount. Furthermore, it adds an extra nonlinearity to the network, which allows for it to learn more complex functions.

Downsampling using 1×1 convolutions has been employed to great success in several milestone architectures: they were used in GoogLeNet's *inception modules* [49] and in ResNet [50] as a means to explicitly control dimensionality.

Let us briefly revisit the usage of pooling layers vs. convolution as a means for downsampling:

- A convolutional layer may learn certain properties, which would otherwise not be captured by pooling layers, by adjusting its weights during the training process.
- On the other hand, learning and storing those weights incurs into greater computational complexity and memory requirements.
- Using large filters and strides for the first convolutional layer, ResNet [50] significantly reduced the dimensionality that deeper layers would have to handle by aggressively compressing spatial information. Other architectures such as SqueezeNet [51] also implemented this.
- FishNet [52] successfully addressed some of ResNet's gradient propagation issues by (among other architectural changes) introducing max-pooling in place of some of the convolutional layers.

2.3.3 Upsampling Layers

The types of layers referred to thus far have addressed CNNs in the context of classification and other related tasks, wherein we wish to reduce dimensionality to a point where we can output a binary decision or a fixed-size categorical prediction. In these contexts, the output layer uses what can be thought of as an abstract, high-level feature representation that the network has learned in order to make a prediction. However, in some applications, we may already have some representation of the features of the data and wish to generate more fine-grained output.

Autoencoders [53] [54] and *generative adversarial networks* [55] are two examples where we wish to generate samples with the same dimensionality as the training data using a learned representation of its features. In these cases, the goal is to translate this coarse representation of the data's features into a denser output which exhibits the same characteristics as the training data.

To perform upsampling, one may use the aptly named *unpooling* operations.

Nearest neighbor unpooling is the upsampling analog to average-pooling. In average-pooling, we take the spatial average of the values within the receptive field of each pooling region. In nearest neighbor unpooling, we take a single value and replicate it to the all target positions of the unpooling region (figure 11).

"Bed of nails" unpooling places the original value in a single, fixed position of the unpooling region. The values of the other positions are filled with zeroes. If we choose to place the original value on the top left, we get a target activation map as shown in figure 12.

Recall that max-pooling takes the maximum of the values in a receptive field. *Max-unpooling* memorizes the positions of the maxima when max-pooling

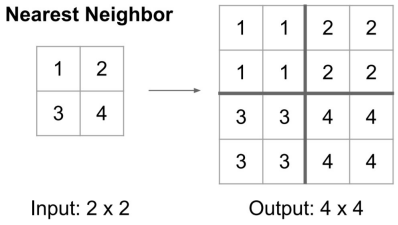

Figure 11: Nearest neighbor unpooling with a 2×2 region and stride 2 [56]

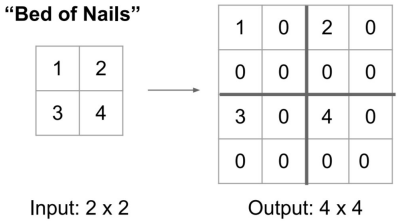

Figure 12: "Bed of nails" unpooling with a 2×2 region and stride 2 [56]

is performed, and places values in those positions. This helps approximate the inverse of the original max-pooling operation [57], but also requires some symmetry between the downsampling and upsampling parts of the network. As in "bed of nails" unpooling, the other positions are filled with zeroes (figure 13).

Unpooling layers, like their downsampling counterparts, have no weights, which we have seen reduces computational complexity when compared to layers with learnable parameters. Learnable upsampling can be achieved using *transpose convolutions*.

Transpose convolutions are also called *fractionally strided convolutions* and *deconvolutions*. However, the latter is a misnomer, since a deconvolution is the inverse operation of convolution in a signal processing context, which transpose convolutions are not. In fact, convolutional layers actually perform something called *cross-relation* [58], but we will stick with the aforementioned nomenclature as it is standard in the literature.

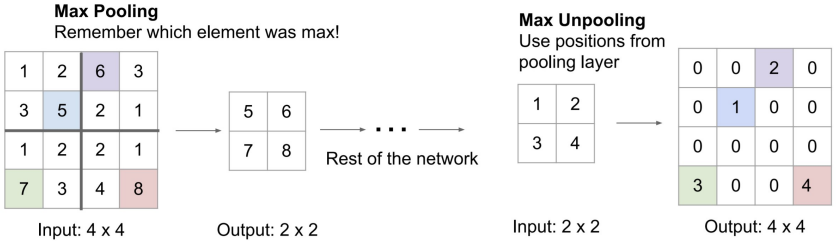

Figure 13: Max-unpooling with a 2×2 region and stride 2 [56]

Transpose convolutional layers have the same hyperparameters as convolutional layers, though they work differently.

In a transpose convolutional layer, each filter learns the mappings from a single value to a set of values, which is the opposite of what convolutional layers do. The filter size, thus, determines how many values a single scalar maps to.

The stride, in this case, is now the ratio between how much we move in the output and how much we move in the input. For example, if a transpose convolutional layer has a stride of 2, that means that for each pixel we shift by in the input, we will shift the filter 2 pixels on the output. Two filters may overlap: if we choose a filter size of 3 and a stride of 2, the third column of the first filter and the first column of the second filter will overlap (see figure 14). In that case, we simply sum the values in the overlapping positions.

Without taking padding into account, the filter size and stride are what determine the spatial size of the output. For example, if we apply a 3×3 filter with stride 2 to a 5×5 feature map, we get a spatial dimension of 11×11 in the output. However, we may want to constrain the dimensions of the output, say, to 10×10. Padding can be used to that effect: in transpose convolutional layers, we may apply padding to both the input and the output so as to force the output to have a certain spatial dimensionality [59] [60].

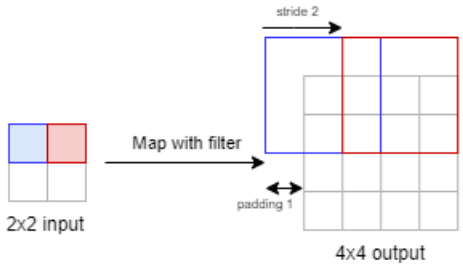

Figure 14: Transpose with 3×3 filter, stride 2, padding 1

One may wonder whether the overlap of multiple receptive fields in the same output region may be troublesome. Indeed, the difference in magnitudes of the output values due to overlapping receptive fields is known to introduce "checkerboard artifacts" in images generated by architectures which use these CNNs [61]. Using a filter size that is divisible by the stride, known as *sub-pixel convolution*, has been successful in tackling this problem [62].

2.4 Generative Adversarial Networks

Generative Adversarial Networks (GANs) are a framework for generative modeling whereby two models engage in a minmax game in order to learn. Those models only need to be differentiable functions, but in practice we use NNs due to their ability to learn complex functions. Since the most common use for GANs is in generating images, we will assume that to be the task at hand.

2.4.1 Generative Modeling

Density estimation is the task of estimating an unobserved probability density function. We can construct an estimation by training a model using samples drawn from the distribution of a probability density function. In particular, generative models address density estimation as they are trained so that they can generate samples that come from the training distribution.

2.4.2 GAN Architecture and Training

The GAN architecture involves a *generator network* G and a *discriminator network* D, which compete in the following manner: given some vector of random noise z as input, the generator G generates a fake image. D receives a batch of both real images from the training data and fake images generated by G. Its task is then to decide whether each image is real or fake. The architecture is shown in figure 15:

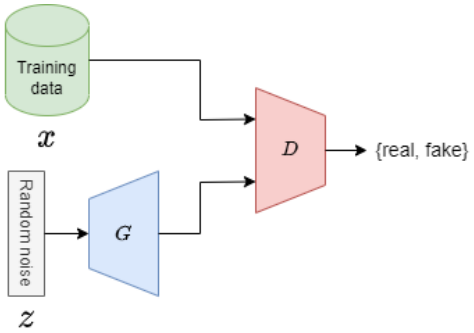

Figure 15: The GAN architecture

Let us denote:

- $G(z)$ as the a fake image created by G.
- $D(x)$ as the probability of D classifying a real sample x as being real.
- $D(G(z))$ as the probability of D classifying a fake sample generated by G as being real.

We can then formulate the GAN training process as a minimax game with the following value function:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{data}}[log(D(\boldsymbol{x}))] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}}[log(1 - D(G(\boldsymbol{z})))], \quad (36)$$

where D tries to make $D(G(z))$ near 0 and G tries to make $D(G(z))$ near 1. We can then write a separate loss function for G and D. The loss for D is just the cross-entropy cost associated with the binary classification problem of telling real from fake samples:

$$\mathcal{L}^{(D)} = -\mathbb{E}_{\boldsymbol{x}\sim p_{data}} logD(\boldsymbol{x}) - \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}} log(1 - D(G(\boldsymbol{z}))) \quad (37)$$

The loss for the generator G is then the negation of the discriminator's loss:

$$\mathcal{L}^{(G)} = -\mathcal{L}^{(D)}. \tag{38}$$

However, since only one of the terms in the formula depends on G, we can further simplify:

$$\mathcal{L}^{(G)} = \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}}\, log(1 - D(G(\boldsymbol{z}))). \tag{39}$$

The authors noted that the above objective for the generator doesn't work well. The generator's loss is plotted in figure 16.

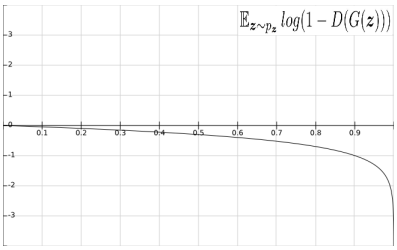

Figure 16: The generator's loss function

We can see that when a sample is likely fake (a small value on the x-axis), the gradient region is relatively flat. This makes it so that at the start of training, when G doesn't produce very good samples, the gradient signal might not be enough to allow the models to converge. A heuristically motivated solution was proposed by the authors, whereby G would have a separate loss:

$$\mathcal{L}^{(G)} = -\,\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}}\, log(D(G(\boldsymbol{z}))), \tag{40}$$

which it tries to *maximize*. Now, instead of minimizing the discriminator payoff, G should maximize the probability that D makes a mistake. In this way, G is less likely to suffer from the vanishing gradient problem (figure 17).

We can see that low-quality samples, produced by G at the beginning of training, will have some gradient to learn from.

2.4.3 Convergence Properties

The equilibrium of this minimax game is a saddle point of the loss of the discriminator D. In particular, finding this saddle point resembles the process of

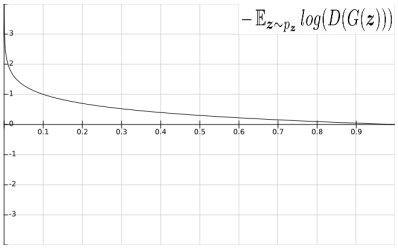

Figure 17: The generator's improved loss function

minimizing the Jensen-Shannon divergence (a measure of the similarity between two probability distributions) between the data and the model.

We can prove that, if we find the equilibrium of the game, then we have retrieved the correct data distribution. In other words, if both G and D have unlimited capabilities, the Nash equilibrium of this game corresponds to G producing samples that are indistinguishable from those in the training data. At that point, D cannot distinguish between the two sources, so it says that a given sample is real or fake with $\frac{1}{2}$ probability.

It is unusual to have a problem with non-convergence when we're dealing with a single loss function. While we may get stuck in local minima, we usually converge to some region where the model performs reasonably well. While most of DL consists of minimizing a single loss function, the main idea of adversarial training is that two players are competing, with each trying to minimize their own cost function. As such, it is possible that GANs may never converge.

During learning, we have D in the "inner loop" and G in the "outer loop":

$$\underset{G}{min}\underset{D}{max}V(G,D) \neq \underset{D}{max}\underset{G}{min}V(G,D). \tag{41}$$

If we minimize G in the inner loop (cf. expression on the right), it will have no incentive to not map all inputs z to the same output, which G considers the most likely to be real. If there is a target distribution with several different modes, the training process is shown to visit one mode after another, instead of visiting all the modes: the generator identifies some mode which the discriminator believes is highly likely, and places all its mass there [63]. For that reason, this particular issue is known as *mode collapse*, and it is why we train D in the inner loop.

2.4.4 Deep Convolutional GANs

In the original GAN architecture, both networks were MLPs. The generator used both ReLU and sigmoid units, and the discriminator used maxout units, along with dropout. *Deep Convolutional Generative Adversarial Networks* (DCGANs) [64] introduced CNNs as an alternative to MLPs for the generator and discriminator. In particular, they:

- Eliminated all deterministic spatial pooling functions (e.g., max-pooling), and replaced them with strided convolutions. The authors argue that this allows the generator and discriminator to learn their own spatial down and up-sampling, respectively.
- Adopted the trend towards eliminating fully connected layers after convolutional layers. Instead, only the last convolutional layer of the discriminator is flattened and connected to a single sigmoid output unit.
- Used batch normalization, which proved critical to get the generator to start learning. However, applying it to all layers resulted in model instability, so it wasn't applied to the generator's output layer and the discriminator's output layer.

Since this successful employment of CNNs in DCGANs, they have become almost standard use in computer vision tasks that leverage GANs.

2.4.5 Conditional GANs

It was suggested in the original GAN paper that both models receive class labels as input during training so as to be conditioned by them, allowing for explicit control on the modes of data being generated. Shortly thereafter, *Conditional Generative Adversarial Networks* (CGANs) were proposed [65].

CGANs are an extension to GANs where both the generator and the discriminator are conditioned on some vector of extra information c:

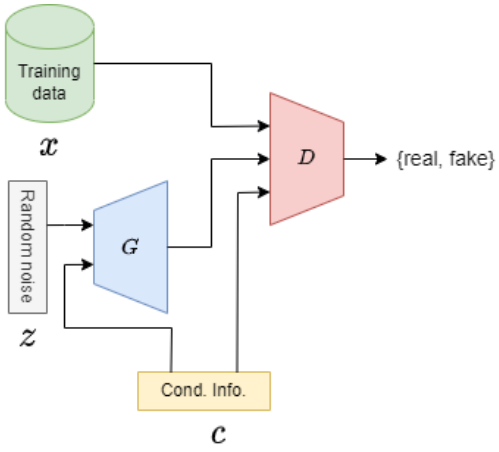

Figure 18: The CGAN architecture

A minimax game with both G and D conditioned on c has the following objective function:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{data}}[log(D(\boldsymbol{x}|\boldsymbol{c}))] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}}[log(1 - D(G(\boldsymbol{z}|\boldsymbol{c})))]. \quad (42)$$

In the case of the generator, adding a conditional vector c just requires doing vector concatenation with the latent noise z. For the discriminator, we add an embedding layer which maps c to the same spatial size as the image, and then add it as a channel to the input image, increasing its depth by 1.

While CGANs allow us to explicitly condition the generator on features that we know about the data, GANs in general make no restrictions as to how the generator uses the noise vector z. This may result in the generator using z in an entangled way. In this aspect, the InfoGAN [66] deserves an honorable mention, as it allows for G to learn semantic features from individual elements of z.

2.5 Related Work

Educators already use automated tools for generating questions for problem sets; like using software to generate algebra multiple choice questions [67]. GANs themselves are also seeing some use in education, such as generating x-ray images of hip fractures to help medical students "train their eye" [68].

Indeed, perhaps one of the most obvious use cases of GANs is in data augmentation. The authors of the original paper had already demonstrated GANs'

ability to generate samples which resembled those in the training set (figure 19), though those had a relatively low resolution of 32×32 pixels.

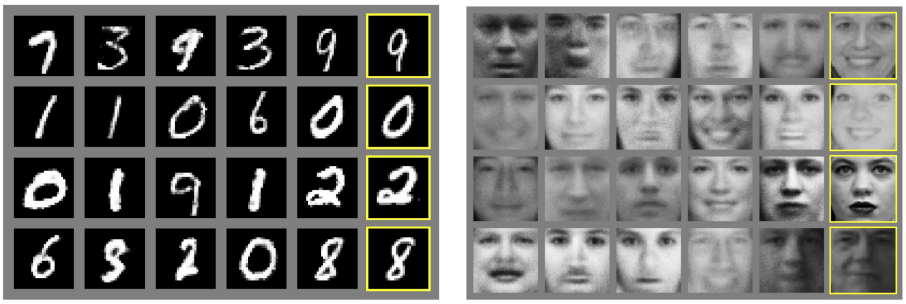

Figure 19: All but the rightmost column show generated samples

A use case of GANs which showcases their ability to scale to larger resolutions is in image super-resolution tasks [69]. These GANs make use of very deep (residual) CNNs to achieve very high resolution images which look subjectively good to human perception. From the super-resolution GAN paper (figure 20), we see that super-resolved images (4x upscaling) are almost indistinguishable from the original ones.

Another task where GANs were successful is that of image translation, a broad term which can be defined as translating a possible representation of a scene into another. The seminal work which introduced the *pix2pix* architecture [70] addressed many image translation tasks (figure 21), including synthesizing photos from label maps, reconstructing objects from edge maps, and colorizing images.

Finally, the CycleGAN [71] was able to learn to transfer the style of an image to another, presenting impressive image translation results (figure 22). This work demonstrated that GAN-based architectures were able to learn even abstract semantic features such as the styles of different painters and changes in landscapes from summer to winter.

For the many impressive applications of GANs since their appearance in 2014 [55], none have yet proposed to solve the problem of generating plots. In the next section, we detail the task we propose to address, as well as our proposed solution and the means with which to evaluate it.

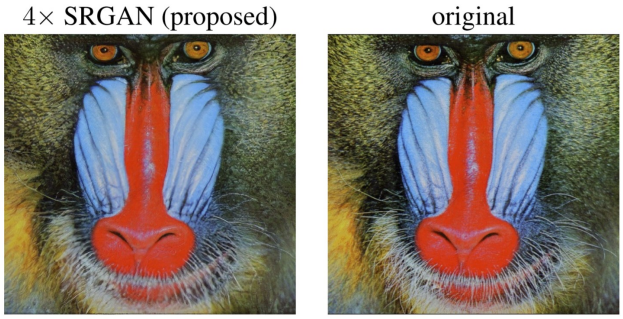

Figure 20: Super-resolved image (left) and original image (right)

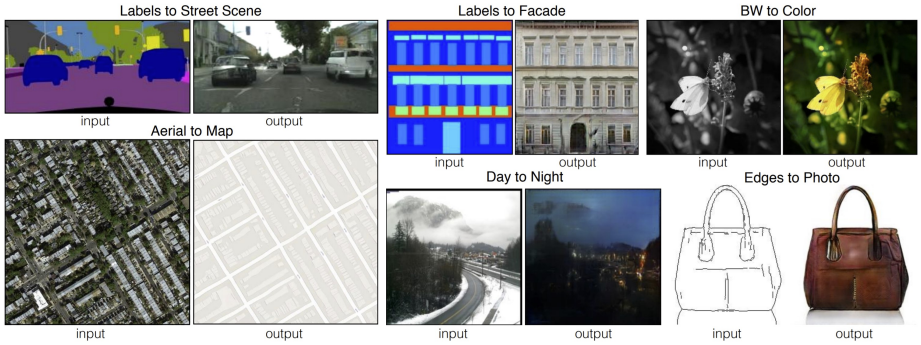

Figure 21: Pix2pix results in some image translation tasks

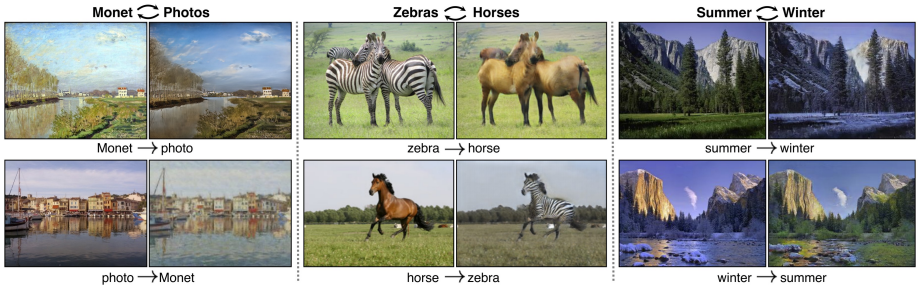

Figure 22: CycleGAN results in some style transfer examples

3 Proposed Approach

In this section we address the specification of the problem to solve, the proposed solution, and the metrics with which to evaluate it.

3.1 Problem Specification

Massive Open Online Courses (MOOCs) have been around for a few years under other names [1], but *The New York Times* called 2012 "The Year of the MOOC" [72]. It saw platforms such as Coursera, Udacity, and edX become mainstream, and associate with education institutions such as universities, which will remain the focus of the discussion herein.

The growth of these platforms was accompanied by that of online learning in general: *blended learning*, which combines face-to-face with online education, has seen an increase in popularity, particularly in higher education [73]. Furthermore, as of 2013, around 70% of education institutions in the U.S.A. claimed that online learning was fundamental to their long-term strategy [74]. Finally, the COVID-19 pandemic has forced a widespread temporary adoption of full online learning [75].

All these factors continuously contribute to the adoption of MOOCs by education institutions, which now face a different paradigm in providing courses. While a discussion of the challenges of online learning is beyond the scope of this document, it is relevant that the informatization of learning systems introduces the opportunity to leverage automated tools to improve the quality of online

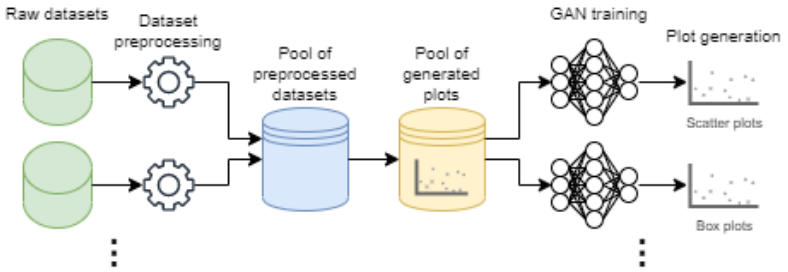

Figure 23: Tool pipeline

courses.

In this work, we consider courses which depend greatly on representing data in plots. Data science, statistics, and machine learning courses are among those which rely on such pictorial representations of data to convey intuition about certain subjects. For example, a scatter plot between two variables helps observe relationships between them. This motivates the development of means whereby educators can automatically generate plots according to certain desired properties. In particular, generating a virtually unlimited amount of plots with characteristics that are relevant to the subject being taught would be of great use: problem sets could be arbitrarily extended so that students were exposed to slightly different representations of the same concept, thus facilitating learning. This use case extends to assessments, where a teacher may want to generate a unique plot for each student, while having control over its properties.

The task we propose to address is, thus, to develop a tool which generates plots according to specific characteristics, which are parametrized by the user. We will focus on scatter plots and box plots, but the developed tool should be easily extensible in order to accommodate other types of charts.

3.2 Solution Specification

The tool we propose makes use of a machine learning model which learns from plots generated from real datasets. For that purpose, we will use the GAN framework, which is the basis of state-of-the-art generative models for images.

3.2.1 Solution Architecture

The developed tool will be a pipeline consisting of the stages depicted in figure 23.

Datasets were collected online and preprocessed into a tractable and common format. In particular, we organized them in comma-separated-values (CSV) files. We will use the Python [76] library Matplotlib [77] to generate the relevant plots for training the GANs. We will also collect the features with which to condition said GANs. By training GANs using a conditional set of features, the goal is for the user to be able to generate new plots with characteristics defined by such features. In other words, an educator would be able to generate plots with properties relevant to a concept, by using different conditional features as

input to the trained GANs. Data preprocessing will be described in the next section.

The GAN implementation will be done in Python3.10.1, using the popular PyTorch [78] machine learning library. The latter offers a powerful framework for deep learning which simplifies implementing models such as neural networks and training them on GPUs for accelerated computation. The architecture of the GAN models will be detailed in a later section.

Once GANs have been trained, we will use them to generate new plots and evaluate them. This evaluation process will also be described in a later section.

3.2.2 Dataset Preprocessing

We chose datasets for classification tasks with real-valued features. We deem restricting variables to being continuous to be a reasonable constraint since it restricts the semantic features to be learned. For example, symbolic variables would introduce columns of data points, which might be too demanding for the model to learn at this time. Examples of datasets to be used are in table 1. For each dataset in the table, values are shown for, respectively:

- The number of samples (i.e., rows) in the dataset.
- The number of features that describe the samples, including classes.
- The number of scatter plots which can be generated using the dataset. It corresponds to the number of 2-combinations among the features which are not classes.
- The number of (single-variable) box plots which can be generated using the dataset. It corresponds to the number of features in the dataset, excluding the classes.

Other than normalizing the datasets into CSV files, non-continuous variables were deleted. We also removed features which were highly correlated, i.e., whose correlation was greater than 0.98. This was to prevent a large number of very similar plots, which could introduce undesired bias into the model.

Recall that in CGANs, we concatenate the normal input with a "conditional vector", which is a set of variables that carry additional information, to form the actual input to the networks.

Similarly, when generating plots between variables, we will store information about each plot. The chosen features ought to be related to relevant features of the created plots, such as data point positioning, density, etc. Examples of such features could be the correlation between two variables, or the variances of each variable.

3.2.3 GAN Architecture

The input to the GANs will be conditioned on a vector of additional information about the plots (cf. CGAN), as detailed in section 2.4.5. Furthermore, we will use CNNs for both the discriminator and generator, as those architectures seem to be the most capable of dealing with image data. As such, we could call the resulting GAN architecture a *Conditional, Deep Convolutional* GAN.

We will comply with the guidelines in the DCGAN paper [64]:

Name	#samples	#features	#scatterplots	#boxplots
Audit Data	776	27	325	26
Polish Companies Bankruptcy	19967	65	2016	64
Liver Disorders	345	7	15	6
Diabetes Database	768	9	28	8
Anuran Calls (MFCCs)	7195	24	210	21
Multiple Features	2000	650	210276	649
MiniBooNE Particle Identification	130064	47	1081	47
Mice Protein Expression	1080	29	378	28
p53 Mutants	31159	5055	12768931	5054
Alcohol QCM Sensor Dataset	125	15	45	10
Sensorless Drive Diagnosis	58509	49	1128	48

Table 1: Datasets used to generate plots

- Remove all pooling layers, using convolution and transpose convolution for downsampling (discriminator) and upsampling (generator), respectively.
- Use batch normalization in both networks.
- Remove fully-connected layers, connecting the last convolutional layer directly to the output layer.
- Use LeakyReLU in all layers of the discriminator except for the last, which uses a single sigmoid unit.
- Use ReLU in all layers of the generator except for the last, which uses a *tanh* unit.

We will use the Adam [36] optimizer, as its use seems to be the universal choice in training GANs [64] [69] [71].

The only preprocessing that shall be done for images is to center them to have a mean of zero, and normalizing them into the range of $[-1, 1]$ - which is also the range of the *tanh* seen in the generator output.

3.3 Solution Evaluation

We expect GANs to have a hard time learning fine details of images. This problem is aggravated when those details have some semantic importance in plots. This is the case for scales, axes names, and other characters in the image containing the plot. For that reason, we propose first training the GANs on images of plots having no descriptive information (i.e., only the axes and data points) and progressively introducing challenging features such as the scale of the variables.

Evaluation of generative models, i.e., how good they are at approximating some data distribution, is very much an open problem [79]. The GAN loss function for each model measures how well they are doing at playing against their opponent, and is not a good measure for image diversity or quality. GANs may be evaluated using quantitative or qualitative metrics. Quantitative methods use some numerical score to evaluate a generated image, whereas qualitative methods involve either subjective human evaluation or a similarity measure (e.g., nearest-neighbor by some metric).

The two most common quantitative metrics for GAN evaluation are the *Inception Score* (IS) [80] and the *Fréchet Inception Distance* (FID) [81]. They both make use of a pretrained CNN, Inception v3 [82], where they get their name from. While these would probably be great choices for evaluating GANs in many other contexts, they are not adequate for this problem. Not only was there no class for charts in the dataset Inception v3 was trained with; there is not enough variation between samples of plots in order for those metrics to be useful.

We will thus turn to qualitative assessment of the generated samples. Manual inspection of samples will be done throughout the training process as a means to detect divergence, mode collapse, and other issues that may come up.

Evaluation of the final results, i.e., after due GAN training and fine tuning, will be done by domain experts: teachers involved in content production for MOOC content.

3.4 Work Schedule

In this section we propose a work schedule for the development of the tool in question and the elaboration of the accompanying dissertation (figure 24). The names corresponding to each of the tasks in the schedule can be seen in table 2.

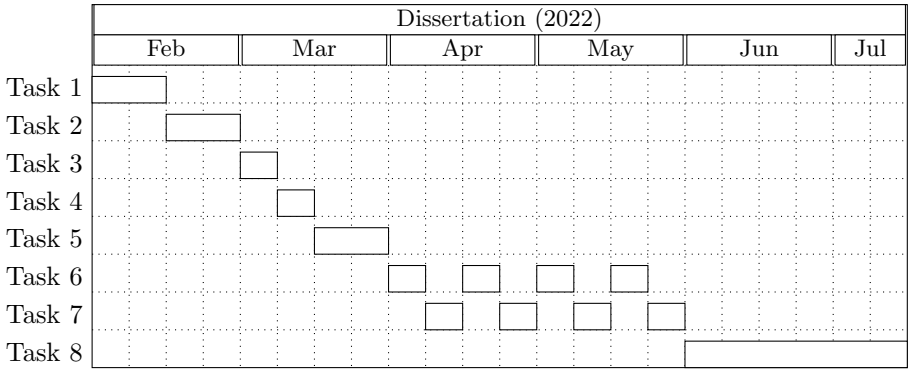

Figure 24: Gantt chart

We will begin by creating a controlled environment where we may conduct the training of GANs (task 1). That includes the specification of software versions and hardware characteristics, such that the experiment may be reproduced as closely as possible if need be.

Task	Task Name
Task 1	Environment setup
Task 2	Data acquisition
Task 3	Plot generation
Task 4	Feature extraction
Task 5	Literature study
Task 6	GAN development
Task 7	Result assessment
Task 8	Document elaboration

Table 2: Definition of each task

Since GANs are very data-hungry [83], we need to make sure we have enough samples with which to train them. We will thus dedicate some time to collect enough datasets and duly preprocessing them (task 2).

After the data are normalized, we may proceed to generate the corresponding scatter plots and box plots (task 3). For each plot, we will also collect the additional information which will be used to condition the GANs (task 4).

With both the data and the development environment ready, we will dedicate some time to searching for relevant literature (task 5). By chance or while working on previous tasks, we may come across relevant work which could be helpful in improving our solution.

The next part is a cycle between developing the GAN models (task 6) and assessing the obtained results (task 7). It is at this time that we may test different configurations of the GAN architecture or introduce more complex tasks (as mentioned in section 3.3).

Finally, a dissertation will be written, explaining the conducted (and related) work and its assessment (task 8).

References

[1] R. Moe, "The brief & expansive history (and future) of the mooc: Why two divergent models share the same name," *Current issues in emerging elearning*, vol. 2, no. 1, p. 2, 2015.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[3] W. S. Mcculloch and W. H. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 1943.

[4] F. Emmert-Streib, Z. Yang, H. Feng, S. Tripathi, and M. Dehmer, "An introductory review of deep learning for prediction models with big data." *Frontiers Artif. Intell.*, vol. 3, p. 4, 2020.

[5] D. O. Hebb, *The Organization of Behavior*. New York: Wiley, 1949.

[6] F. Rosenblatt, "The perceptron, a perceiving and recognizing automaton project para," Cornell Aeronautical Laboratory, Tech. Rep., 1957.

[7] ——, "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958. [Online]. Available: https://doi.org/10.1037/h0042519

[8] M. Minsky and S. Papert, *Perceptrons*. Cambridge, MA: MIT Press, 1969.

[9] D. E. Rumelhart and J. L. Mcclelland, Eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*. Cambridge, MA: MIT Press, 1986.

[10] J. L. Mcclelland, D. E. Rumelhart, and G. E. Hinton, "The appeal of parallel distributed processing," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, D. E. Rumelhart and J. L. Mcclelland, Eds. Cambridge, MA: MIT Press, 1986, pp. 3–44.

[11] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, pp. 193–202, 1980.

[12] A. G. Ivakhnenko, "The group method of data of handling; a rival of the method of stochastic approximation," *Soviet Automatic Control*, vol. 13, pp. 43–55, 1968.

[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[14] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986. [Online]. Available: http://dx.doi.org/10.1038/323533a0

[15] ——, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, D. E. Rumelhart and J. L. Mcclelland, Eds. Cambridge, MA: MIT Press, 1986, pp. 318–362.

[16] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen. diploma thesis, institut für informatik, lehrstuhl prof. brauer, technische universität münchen," 1991.

[17] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.

[18] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[19] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the 5th Annual Workshop on Computational Learning Theory (COLT '92), July 27-29, 1992, Pittsburgh, PA, USA*, D. Haussler, Ed. ACM Press, New York, NY, USA, 1992, pp. 144–152. [Online]. Available: http://doi.acm.org/10.1145/130385.130401

[20] C. Cortes and V. Vapnik, "Support-vector networks." *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[21] G. E. Hinton, S. Osindero, and Y. W. Teh, "A fast learning algorithm for deep belief nets." *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, 2006.

[22] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks." in *NIPS*, B. Schölkopf, J. C. Platt, and T. Hofmann, Eds. MIT Press, 2006, pp. 153–160.

[23] M. Ranzato, C. S. Poultney, S. Chopra, and Y. LeCun, "Efficient learning of sparse representations with an energy-based model." in *NIPS*, B. Schölkopf, J. C. Platt, and T. Hofmann, Eds. MIT Press, 2006, pp. 1137–1144.

[24] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals, and Systems*, vol. 2, pp. 303–314, 1989.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

[26] A. L. Maas, A. Hannun, and A. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *International Conference on Machine Learning (ICML-13)*, 2013.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*. IEEE, Dec. 2015. [Online]. Available: https://doi.org/10.1109/iccv.2015.123

[28] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)." in *ICLR (Poster)*, Y. Bengio and Y. LeCun, Eds., 2016.

[29] A. Y. Ng, "Feature selection, l1 vs. l2 regularization, and rotational invariance," in *Twenty-first international conference on Machine learning - ICML '04*. ACM Press, 2004. [Online]. Available: https://doi.org/10.1145/1015330.1015435

[30] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[31] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

[32] P. Luo, X. Wang, W. Shao, and Z. Peng, "Towards understanding regularization in batch normalization." in *ICLR (Poster)*. OpenReview.net, 2019.

[33] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How does batch normalization help optimization?" in *Proceedings of the 32nd international conference on neural information processing systems*, 2018, pp. 2488–2498.

[34] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Networks*, vol. 12, no. 1, pp. 145 – 151, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608098001166

[35] J. C. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization." in *COLT*, A. T. Kalai and M. Mohri, Eds. Omnipress, 2010, pp. 257–269. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/1953048.2021068

[36] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, cite arxiv:1412.6980 Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[37] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, J. Fürnkranz and T. Joachims, Eds., 2010, pp. 807–814.

[38] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, Jan. 2015. [Online]. Available: https://doi.org/10.1016/j.neunet.2014.09.003

[39] D. H. Hubel and T. N. Wiesel, "Receptive fields of single neurons in the cat's striate cortex," *Journal of Physiology*, vol. 148, pp. 574–591, 1959.

[40] ——, "Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex," *Journal of Physiology (London)*, vol. 160, pp. 106–154, 1962.

[41] O. Abdel-Hamid, A. rahman Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition." *IEEE ACM Trans. Audio Speech Lang. Process.*, vol. 22, no. 10, pp. 1533–1545, 2014.

[42] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015.

[43] A. Borovykh, S. M. Bohté, and C. W. Oosterlee, "Conditional time series forecasting with convolutional neural networks," *arXiv preprint arXiv:1703.04691*, 2017.

[44] B. Chidester, T. Zhou, M. N. Do, and J. Ma, "Rotation equivariant and invariant neural networks for microscopy image analysis," *Bioinformatics*, vol. 35, no. 14, pp. i530–i537, Jul. 2019. [Online]. Available: https://doi.org/10.1093/bioinformatics/btz353

[45] M. Yani, S. Irawan, and C. Setianingsih, "Application of transfer learning using convolutional neural network method for early detection of terry's nail," *Journal of Physics: Conference Series*, 2019, image available under the Creative Commons license 3.0.

[46] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," in *Artificial Neural Networks – ICANN 2010*. Springer Berlin Heidelberg, 2010, pp. 92–101. [Online]. Available: https://doi.org/10.1007/978-3-642-15825-4_10

[47] C.-Y. Lee, P. Gallagher, and Z. Tu, "Generalizing pooling functions in CNNs: Mixed, gated, and tree," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 863–875, Apr. 2018. [Online]. Available: https://doi.org/10.1109/tpami.2017.2703082

[48] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

[49] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Jun. 2015. [Online]. Available: https://doi.org/10.1109/cvpr.2015.7298594

[50] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Jun. 2016. [Online]. Available: https://doi.org/10.1109/cvpr.2016.90

[51] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size," 2016, cite arxiv:1602.07360Comment: In ICLR Format. [Online]. Available: http://arxiv.org/abs/1602.07360

[52] S. Sun, J. Pang, J. Shi, S. Yi, and W. Ouyang, "Fishnet: A versatile backbone for image, region, and pixel level prediction." in *NeurIPS*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 762–772.

[53] D. H. Ballard, "Modular learning in neural networks." in *AAAI*, K. D. Forbus and H. E. Shrobe, Eds. Morgan Kaufmann, 1987, pp. 279–284.

[54] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, Jul. 2006. [Online]. Available: https://doi.org/10.1126/science.1127647

[55] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014, cite arxiv:1406.2661. [Online]. Available: http://arxiv.org/abs/1406.2661

[56] F.-F. Li, J. Johnson, and S. Yeung, "Stanford cs231n - convolutional neural networks for visual recognition, lecture 11," 2017, published under the MIT license.

[57] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Computer Vision – ECCV 2014*. Springer International Publishing, 2014, pp. 818–833. [Online]. Available: https://doi.org/10.1007/978-3-319-10590-1_53

[58] W. Shi, J. Caballero, L. Theis, F. Huszar, A. P. Aitken, C. Ledig, and Z. Wang, "Is the deconvolution layer the same as a convolutional layer?" *CoRR*, vol. abs/1609.07009, 2016.

[59] Conv2DTranspose of the Tensorflow API.

[60] ConvTranspose2d of the PyTorch API.

[61] A. Odena, V. Dumoulin, and C. Olah, "Deconvolution and checkerboard artifacts," *Distill*, 2016. [Online]. Available: http://distill.pub/2016/deconv-checkerboard

[62] W. Shi, J. Caballero, F. Huszar, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang, "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Jun. 2016. [Online]. Available: https://doi.org/10.1109/cvpr.2016.207

[63] L. Metz, B. Poole, D. Pfau, and J. Sohl-Dickstein, "Unrolled generative adversarial networks." *CoRR*, vol. abs/1611.02163, 2016.

[64] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2015, cite arxiv:1511.06434Comment: Under review as a conference paper at ICLR 2016. [Online]. Available: http://arxiv.org/abs/1511.06434

[65] M. Mirza and S. Osindero, "Conditional generative adversarial nets," 2014, cite arxiv:1411.1784. [Online]. Available: http://arxiv.org/abs/1411.1784

[66] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever, and P. Abbeel, "Infogan: Interpretable representation learning by information maximizing generative adversarial nets," 2016, cite arxiv:1606.03657. [Online]. Available: http://arxiv.org/abs/1606.03657

[67] A. Moura Santos, P. Alex, R. Santos, F. Dion, P. Duarte, and C. Lisboa, "On-line assessment in undergraduate mathematics - an experiment using the system cal for generating multiple choice questions," 06 2002.

[68] S. G. Finlayson, H. Lee, I. S. Kohane, and L. Oakden-Rayner, "Towards generative adversarial networks as a new paradigm for radiology education." *CoRR*, vol. abs/1812.01547, 2018.

[69] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Cunningham, A. Acosta, A. P. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, "Photo-realistic single image super-resolution using a generative adversarial network." in *CVPR*. IEEE Computer Society, 2017, pp. 105–114.

[70] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," 2016, cite arxiv:1611.07004Comment: Website: https://phillipi.github.io/pix2pix/, CVPR 2017. [Online]. Available: http://arxiv.org/abs/1611.07004

[71] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks." *CoRR*, vol. abs/1703.10593, 2017.

[72] "The new york times: The year of the mooc," https://www.nytimes.com/2012/11/04/education/edlife/massive-open-online-courses-are-multiplying-at-a-rapid-pace.html, accessed: 2022-01-09.

[73] J. S. Drysdale, C. R. Graham, K. J. Spring, and L. R. Halverson, "An analysis of research trends in dissertations and theses studying blended learning," *The Internet and Higher Education*, vol. 17, pp. 90–100, 2013.

[74] I. E. Allen and J. Seaman, *Changing course: Ten years of tracking online education in the United States.* ERIC, 2013.

[75] S. Dhawan, "Online learning: A panacea in the time of covid-19 crisis," *Journal of Educational Technology Systems*, vol. 49, no. 1, pp. 5–22, 2020.

[76] Python, a general-purpose programming language.

[77] Matplotlib, a Python library for visualization.

[78] PyTorch, a Python library for machine learning.

[79] A. Borji, "Pros and cons of gan evaluation measures." *CoRR*, vol. abs/1802.03446, 2018.

[80] T. Salimans, I. J. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans." in *NIPS*, D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, Eds., 2016, pp. 2226–2234.

[81] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a local nash equilibrium." in *NIPS*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 6626–6637.

[82] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015, cite arxiv:1512.00567. [Online]. Available: http://arxiv.org/abs/1512.00567

[83] S. Zhao, Z. Liu, J. Lin, J.-Y. Zhu, and S. Han, "Differentiable augmentation for data-efficient gan training." *CoRR*, vol. abs/2006.10738, 2020.