GAN Project 2022-23
Joseph Perry

Contents

---

1 Introduction

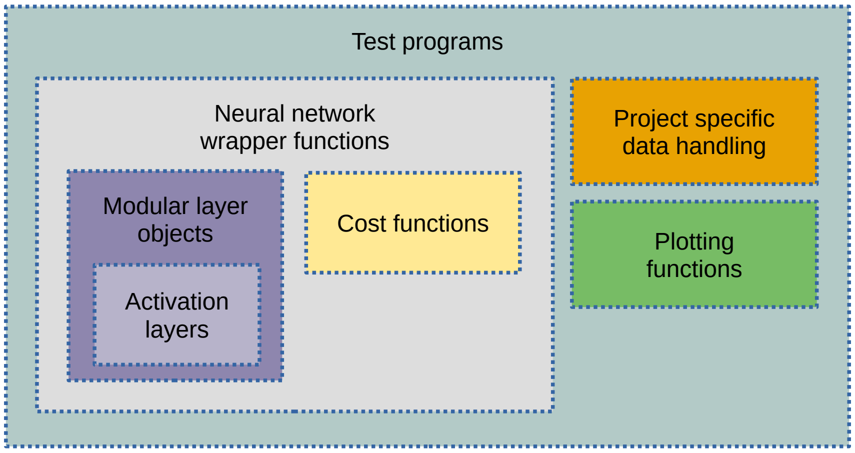


Figure 1. Program structure

For this project I have written a small library of neural network functions for implementing Generative Adversarial Networks. (GANs) This library consists of a number of modular neural network layers, made functional as neural networks by a module of wrapper functions. The wrapper uses a module of cost functions to measure error in order to update the layer parameters with stochastic gradient descent. The data handling and plotting modules are used to encapsulate common data and plotting operations used in the test programs.

1.1 Library modifications

The base code for the modular layers is taken from TheIndependentCode's neural network library. **[1]** I chose this library as a base because of its modularity and the excellent documentation available in the associated video series, **[2]** which was invaluable for developing an understanding of the convolutional layers used in the program. Thanks to the modularity and clear documentation of the library I was able to adapt and add-to the code so that it was more suited for this project. The tests for the adapted code can be found in the source code for this project.

The changes to the library:

- Modified backwards method of trainable layers so that calculated gradients are summed instead of immediately applied so that layers can adjust with the average gradient of a batch, rather than with gradient of each item.
- Implemented a Transposed Convolutional layer by inheriting the methods of the existing Convolutional layer and switching the convolution types in the propagation methods.
- Implemented a Max pooling layer with variable input size and pool size.
- Implemented a Padding layer with variable input size, edge padding and pixel padding.
- Implemented a ReLU activation function layer with variable input size.
- Implemented a Leaky-ReLU activation function layer with variable input size.

1.2 Testing methods

The optimization strategy used to train the neural networks is stochastic gradient descent. To make sure the optimizer was working properly, I tested optimising neural networks for a constant input and target activation, plotting the average cost and output activation at each training step with the assumption that a graph showing decreasing cost and an output activation approaching it's target would indicate a working training strategy. I optimized mockup discriminator and generator networks in this way to test that both components could specialise. Performance on classification tasks was measured using classification accuracy and cross validation scores, first on a simple data then on **MNIST**. **[3]**

To test the GANs I first tested their ability at capturing simple generative targets, the simplest was a single two-dimensional point intended to illustrate how the generated estimates change over the course of training. The next targets used were **true_image**, a single 10x10 pixel monochrome image of a square with a diagonal line through it, equivalent to a single multidimensional point and **digits**, a set of ten 10x10 pixel monochrome images of the digits zero to nine, equivalent to ten multidimensional points. With this data I tested a fully-connected GAN model and a convolutional GAN model with several different training parameters, recording costs, activations and intermediate generated images for each configuration.

1.3 Details

Software

Name	Version	Description
python	**3.10.8**	Implementation language
numpy	**1.22.4**	Python matrix arithmetic library
matplotib	**3.6.1**	Python plotting library
scipy	**1.9.0**	Python scientific processing library
visualkeras	**0.0.2**	Visualisation library for keras networks
ffmpeg	**4.4.3**	Linux video processing tool
bash	**5.1.16**	Linux shell used for scripting
NN-SVG	**1.0**	Web-based neural network visualisation tool

External datasets

Location	Description
data/mnist/	Local MNIST data

Figures

Location	Description
figures/data/	Representations of **true_image** and **digits**
figures/external/	Figures from external sources
figures/layers/	Section 4 figures
figures/networks/	Neural network testing
figures/gans/	GAN testing
figures/gans/igivids/	Videos from GAN testing
figures/gans/other/	Extra development figures from GAN testing
figures/gans/pointgans/	Figures from 2D-point generating GAN
figures/gans/visualise/	Visualkeras GAN representations

Library code

Location	Description
library/activations.py	Activation function layers
library/costs.py	Cost functions
library/handling.py	Data handling operations
library/layers.py	Neural network layers
library/networks.py	Neural network layer wrapper functions
library/plotting.py	Data plotting operations
library/to_video.sh	Bash script for converting PNGs in **library/igis/** into video with **ffmpeg**

Testing code

Location	Description
library/test_afunc.py	Plots of the activations functions
library/test_conv.py	Example plots of different convolutions using scipy
library/test_mpl.py	Tests for the MaxPooling2D layer
library/test_padding.py	Tests for the ZeroPadding2D layer
library/test_gans.py	Tests of different GAN models for different parameter configurations and datasets
library/test_networks.py	Tests for different neural network functionalities
library/test_pointgans.py	Tests for 2D-point generating GAN
library/test_visualise.py	Visualkeras visualisations of the GAN models used in **test_gans.py**

2 Developing an understanding of neural networks

[4] [5] Neural networks, or just networks, are a type of composite function used in machine learning. The network function is the composition of an arbitrary number of functions called layers. Typically, each layer has a number of parameters that are involved with the transformation of input to output. The changes to these parameters in response to error can be calculated using a process called backpropagation, which begins with a metric of error called a cost function and changes each parameter proportionally to how much it was responsible for the magnitude of the error. These changes are then applied in an iterative process called gradient descent, which aims to minimise the total error measured by the cost function.

2.1 Forward propagation

The forward propagation of the input **x** through the composite network function **N** for a network of **n** layers $\mathbf{L_{1 \ldots n}}$ can be represented as:

$$\mathbf{N(x) = (L_n \circ L_{n-1} \ldots \circ L_2 \circ L_1) (x)}$$

Where each layer function is also a function of its own parameters.

2.2 Measuring error using cost functions

The error, also referred to as cost, of the network's output is evaluated against a target output using functions called cost functions. The value of cost is small when the network output is similar to the target output and large when the network output is dissimilar to the target output. The **cost** of a network output **N(x)** against a target network output **y** with the cost function **C**, can be represented as:

$$\mathbf{cost = C(N(x), y)}$$

Where the network function **N** is also a function of its own parameters.

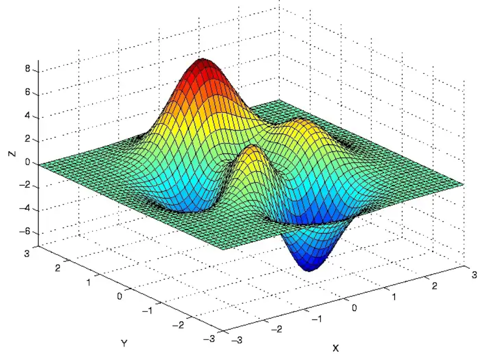
2.3 Backpropagation and Gradient Descent

Backpropagation is the process of calculating for each parameter in the network, the amount of effect it had on the value of cost, or more specifically, the derivative of **C(N(x), y)** with respect to each parameter in the network.

To find these derivatives, the chain rule is used to differentiate the composite function $\mathbf{C((L_n \circ L_{n-1} \ldots \circ L_2 \circ L_1) (x), y)}$ with respect to the parameters of each layer, one layer at a time, moving backwards through the network from the output layer $\mathbf{L_n}$ to the input layer $\mathbf{L_1}$.

The values of the parameters can be thought as a position in a multidimensional space called parameter space. The gradient of the cost function given the parameter values is therefore the slope of the cost function topology at that position in parameter space.

By finding the gradient of the cost function at the current position in parameter space, it is possible to adjust the parameters of the network iteratively in the downhill direction of the gradient until the minimum value of cost is found. This iterative process is called gradient descent.



[6] Figure 2. Three dimensional representation of the topology of a cost function with two parameters, where the z axis represents the cost value and the x-y plane represents the parameter space.

2.4 Shuffle-Split and Stochastic Gradient Descent

When minimising the cost value for a full dataset, the gradient applied at each gradient descent step is the average of the gradients calculated for each labelled item in the dataset. For large datasets it is often too computationally expensive to calculate the gradient for the whole dataset at each gradient descent step. To solve this, the items in the dataset are first shuffled, then split into a number of batches, with the hope that each batch captures roughly the same data distribution as the whole dataset. When training with stochastic gradient descent, the network is updated for one step for each of these batches, taking less precise, but less computationally expensive steps.

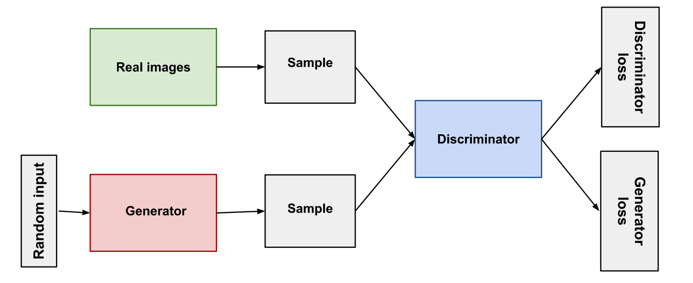
3 Developing an understanding of GANs

GANs are a type of generative model, typically composed of two neural networks; a discriminator network and a generator network. The aim of training a GAN is to find an equilibrium between the generator and discriminator networks, called a Nash equilibrium, where the generator network is able to produce samples that are visually similar to those in the target dataset. In this report I will discuss GANs in the context of image generation, but it is important to remember that the target data could theoretically take any form, providing the GAN is constructed appropriately for the task.

3.1 Generator and Discriminator

[7] The discriminator network's aim is to correctly classify data as either real (a sample from the target dataset) or fake (a generated sample) with a probability score between **0** and **1**, where **1** represents certainty that a sample is real and **0** represents certainty that a sample is fake.

The generator network's aim is to generate data samples which are similar enough to samples from the target dataset that the discriminator is unable to differentiate between them. The inputs to the generator are random vectors which can be thought of as random points in a multidimensional space called latent space.



[8] Figure 3. Diagram of the GAN model from the google developers advanced course on GANs.

3.2 Training

The goal of GAN training is to learn a mapping from latent space to a set of outputs with a similar probability distribution to that of the dataset. To achieve this, the model must be trained to identify the defining features of the dataset and associate them with coordinates in latent space through an alternating training process between the discriminator and generator networks, where the two networks are put together as adversaries.

In the discriminator training phase the discriminator network is presented with **m** samples from the dataset and **m** generated samples. The generated samples are produced by the generator network's transformation of **m** samples of noise. The dataset samples are labelled with probability **1** for real, whilst generated samples are labelled with probability **0** for fake. The discriminator is then tasked with binary classification of these two categories, updating it's trainable parameters based on the errors it made in it's classifications.

In the generator training phase the generator network is presented with **2m** samples of noise which it transforms into **2m** generated samples. These samples are all labelled **1** for real, then classified by the discriminator network. The error is backwards propagated first through the discriminator then through the generator, updating the trainable parameters of only the generator network, based on the errors that it made in fooling the discriminator.

[9] The training process guides the probability distribution of the generated samples (the generative distribution) using the judgement of the discriminator (the discriminative distribution) until the generative distribution is indistinguishable from the probability distribution of the dataset, (the data generating distribution) where the training converges.

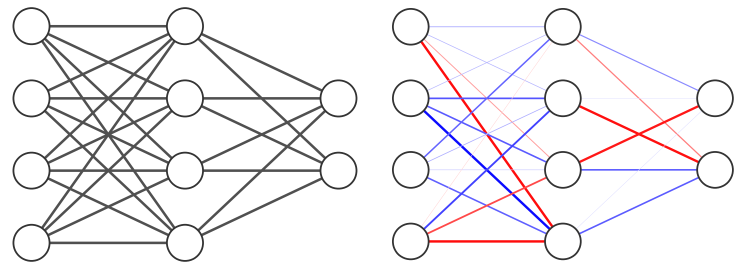
4 Developing an understanding of layer types

4.1 Dense layers

[10] A dense layer, sometimes called a fully connected layer, is a function with an arbitrary number of inputs called input nodes represented as the column vector **x**.

Each of these input nodes has a weight value associated with each of an arbitrary number of output nodes. These weights are represented in the matrix **W**.

The activation of a single output node in a fully connected layer is the weighted sum of all the activations of the input nodes, plus the output node's bias value. The biases of the all outputs are represented as the column vector **b**.



[11] Figure 4. NN-SVG visualisation of the connections between the inputs and outputs (left) and the associated weights of each connection (blue: positive, red: negative) (right) in two connected dense layers, $\mathbf{L_2(L_1(x))}$.

The fully connected layer function **L** with input **x**, weights **W** and biases **b** can be represented as: $\mathbf{L(x) = x \cdot W + b}$

4.2 Activation layers and Vanishing Gradient

[12] Activation layers apply a non-linear transformation, or activation function, to each component of their input. Without a non-linear relationship between the otherwise linear layer functions, the network could be simplified to a single linear layer, which would not allow as complex of a representation as a network of layers separated by non-linearities.

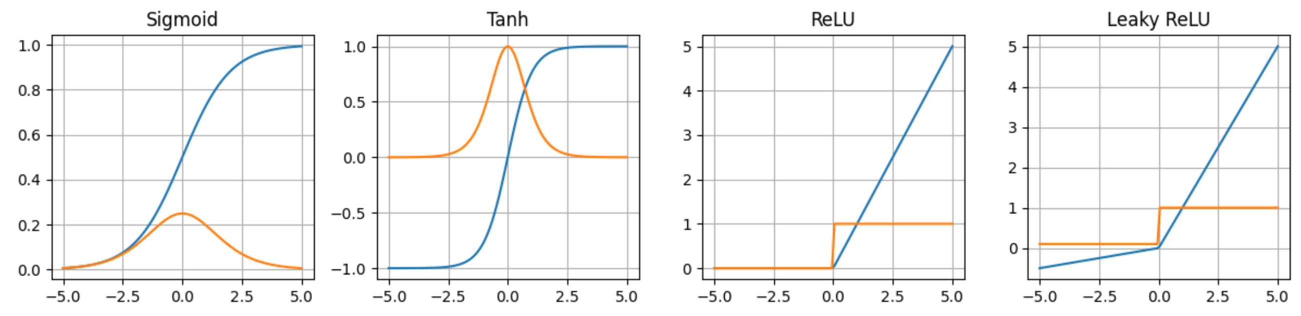


Figure 5. Plots of different activation functions (blue) [Logistic, Tanh, ReLU, leaky ReLU] and their derivatives. (orange)

Function name	Alternative function name	Expression	Interval notation
Sigmoid	Logistic	$\sigma(x) = 1 / (1 + e^{-x})$	[0, 1]
Tanh	Hyperbolic Tangent	tanh(x) = sinh(x) / cosh(x)	[-1, 1]
ReLU	Rectified Linear Unit	ReLU(x) = if x > 0, x, else 0	[0, +∞]
Leaky ReLU	Leaky Rectified Linear Unit	LReLU(x) = if x > 0, x, else x*a where 0<a<1	[-∞, +∞]

[13] The first two functions, Sigmoid and Tanh, are called saturating non-linearities. This means that either side of a certain value range, the output value of the function will not change meaningfully with larger negative or positive values. The second two functions, ReLU and Leaky ReLU, are called non-saturating non-linearities, which means that for increasingly large input values, the output will continue to reflect the magnitude of the input.

Vanishing gradient is an issue that can occur when using saturating non-linearities because the rate of change of the function slows to zero as the function saturates. This means that during backpropagation, any parameter derivatives that are a product of the non-linearity derivative (because of the chain rule), will also slow to zero and vanish if the activation function ahead of them is saturated. For non-saturating non-linearities, the problem of vanishing gradient is not such an issue because the rate of change of the function is constant across the value range.

Considering the concept of a cost function topology, the concept of vanishing gradient is analogous to the situation where the cost function topology at the position in parameter space is too flat for the gradient descent step to have a clear direction.

4.3 Convolutional layers and Transposed convolutional layers

[14] [15] [16] Convolutional and Transposed Convolutional layers are layers that are particularly useful in image processing. They perform an operation called a convolution between an input image and a number of small matrices called convolution kernels. The result is a feature map indicating where in the image certain features are most prominent.

A convolutional layer performs an operation called a valid convolution between convolution kernels and an input image, whilst a transposed convolutional layer performs a full convolution.

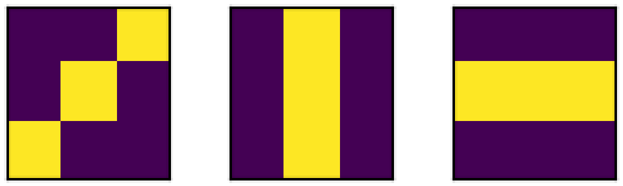


Figure 6.1. 3x3 convolution kernels able to detect diagonal, vertical and horizontal features respectively.

In both convolution operations, the kernel is first rotated 180° then moved across the image incrementally, left-to-right-downwards, where for each position, the pixel values in the image which lie underneath the kernel are multiplied by the corresponding values in the kernel. The multiples are then summed and the sum is stored in an output called a feature map at a position corresponding to the position of the kernel over the image.

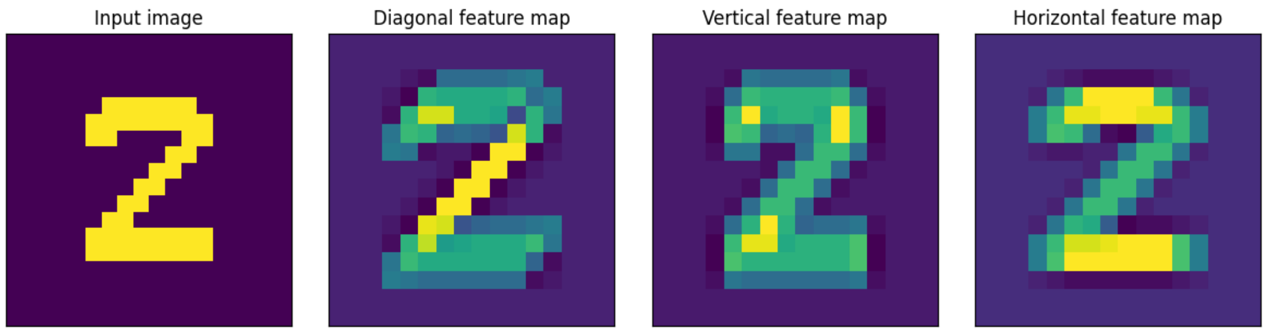


Figure 6.2. An input image and the feature maps resulting from its convolution with the figure 6.1 kernels.

The difference between valid and full convolution is the way the kernel behaves when moving across the borders of the image. In a valid convolution, the edges of the kernel are strictly bound within the image borders so that each corner of the kernel never moves beyond the same corner of the image, whilst In a full convolution, the kernel begins at the point where it would first intersect the image if it were traversing all of 2D space left-to-right-downwards.

A valid convolution will normally produce an output resolution smaller than the input resolution, whilst a full convolution will normally produce an output resolution larger than the input resolution. The resolution of the kernel, stride and padding will also effect the output resolution, by effecting the number of possible kernel positions over the image during convolution.

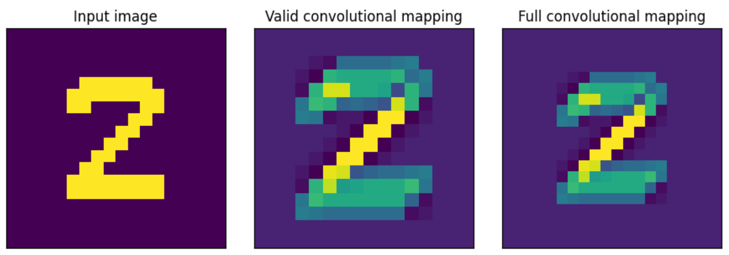


Figure 6.3. An input image and the feature maps produced by valid convolution and full convolution with the figure 6.1 diagonal kernel.

In valid convolution, larger kernels will reduce the output resolution, whilst in full convolution larger kernels will increase the output resolution.

Stride is the step size that the kernel takes when moving across the image during convolution, increasing it will decrease the output resolution.

Padding is a modification to the input image that inserts values either in-between or around the input pixels, effecting the output resolution simply by making the input image larger.

These examples describe convolution with regards to a single-channel image, but a convolutional layer can have any number of input channels. In such cases, each kernel has an equal number of channels and the resulting feature map is the sum of the convolutions of the corresponding channels

4.4 Max pooling layers

Max pooling layers work to filter the input so that only the most important values remain in the output. They work by dividing the input into a grid of rectangles, or pools, and taking the maximum value in each pool as the output.

[17] Because the max pool layer function is essentially to multiply maximum values by 1 and others by 0, the differential of the output is 1 for inputs that were maximum and 0 for inputs that were not.

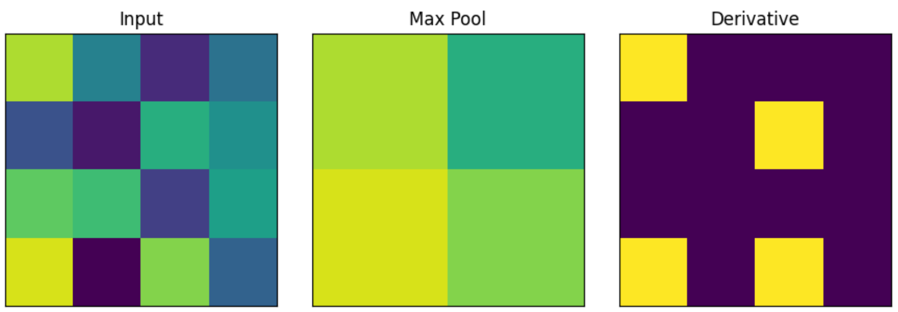


Figure 7. The max pool of a 4x4 input of values between 0 and 1, with 2x2 sized pools produces a 2x2 output. The derivative is 1 for max values and 0 for non-max values.

4.5 Reshape layers and Padding layers

In the network implementation, there are also types of layers that are responsible for modifying the characteristics of data as it passes through the network. Two such layers are reshape layers and padding layers. Reshape layers are responsible for altering the shape of the matrices that pass through them. They are most commonly used as an adapter between convolutional layers and dense layers, which require the reshape of a 3d matrix into a column vector.

Padding layers are used to add a pattern of zeros in or around the pixels of an input image, usually preceding a convolutional or transposed convolutional layer to implement the convolutional padding hyperparameter.

Padding is an alternative to varying convolution kernel size for controlling the resolution of the output image for both types of convolutional layer that effects the output resolution by altering the resolution of the input.

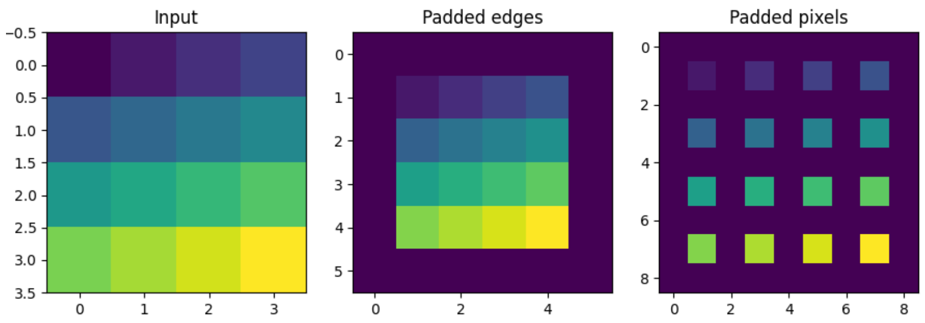


Figure 8. The visual appearance an input image (left) padded with one pixel of edge padding (centre) and one pixel of pixel padding. (right)

4.6 Cost functions

[18] The derivatives of cross-entropy and mean squared error (MSE) cost functions have different magnitudes in response to the amount of error. The cross-entropy cost function is more sensitive to large errors than the mean squared error (MSE) cost function, which in most cases leads the optimizer to a solution faster than MSE.

These differences are due to the differences in the derivatives of the cost functions. The magnitude of the derivative of the cross-entropy cost function is relatively small when the error is small and becomes larger as the error increases, whilst the magnitude of the derivative of the MSE cost function is proportional to the error.

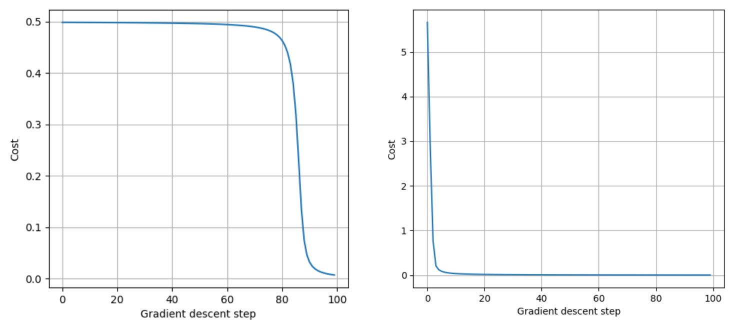


Figure 9. Networks optimising from positions with high initial error using MSE (left) and cross-entropy. (right)

5 Neural network testing

5.1 Optimizer testing

To test the SGD optimizer, I optimized two different network configurations intended to represent the GAN components on a constant input and target activation, meaning at each optimization step the input to the network and the target output remained the same. In these tests, the constant input is a random set of values, but a selected set of constant values would also produce the same results.

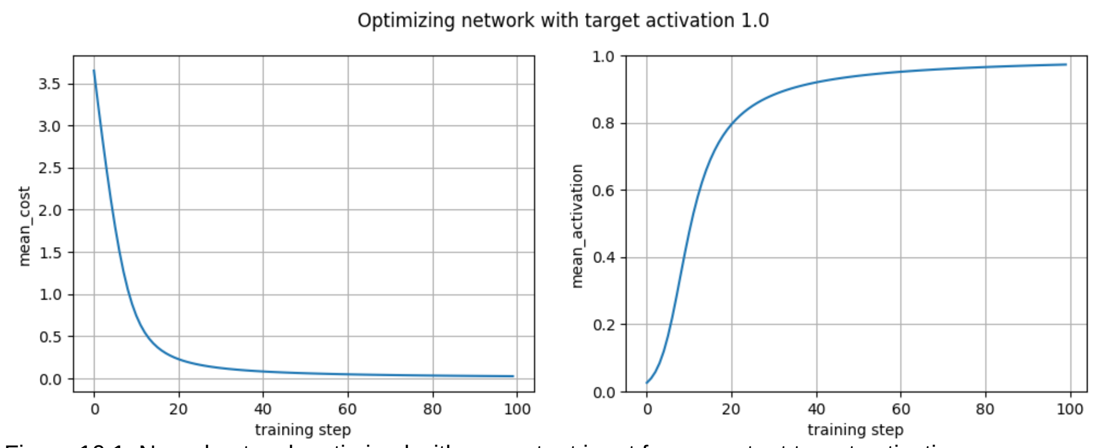


Figure 10.1. Neural network optimised with a constant input for a constant target activation.

Updating the discriminator component of a GAN is the same as updating a normal classifier network. Data is forward propagated through the network, the error is calculated against a target label and the parameters are adjusted in response to error using backpropagation. Plotted in figure 10.1 are the results of a network trained in this way with a constant input and a target activation of **1**. This figure represents a successful test because the cost is decreasing and the output activation is approaching its target.

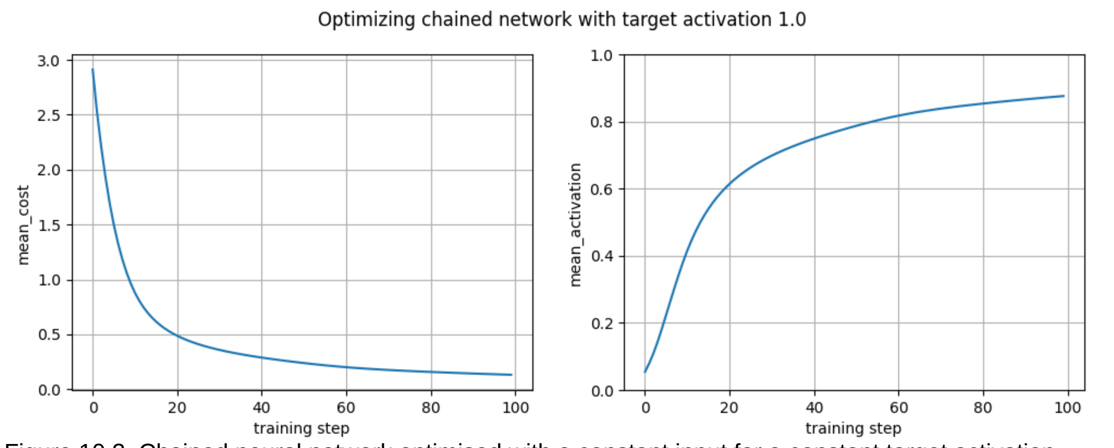


Figure 10.2. Chained neural network optimised with a constant input for a constant target activation.

Updating the generator component of a GAN requires forward propagating inputs through the generator then through the discriminator, but updating the parameters of only the generator during backpropagation. This means backpropagating through the discriminator without parameter updates then through the generator with parameter updates. In other words, updating only **a** in the network **a -> b**, where **a** is referred to as the chained network and **b** is referred to as the static network. Plotted in figure 10.2 are the results of optimising only **a** in the network **a->b** for a constant input and target activation of **1**. This figure represents a successful test because the cost is decreasing and the output activation is approaching its target.

5.2 Classifier evaluation

To evaluate the classification performances of the neural networks I first used classification accuracy, then cross-validation accuracy metrics to measure classification performances of networks trained on labelled **digits** and **MNIST** data.

The classification accuracy score calculates the ratio of correct to incorrect predictions that a model makes when classifying a dataset of x,y pairs. The classification accuracy score can be expressed as:

$$\textbf{classification accuracy} = \frac{\textbf{correct predictions}}{\textbf{total predictions}}$$

For figure 11.1 I used **digits** as the target x values and the numerical values of each digit as the target y values. The goal was to train a dense network until it found a fully accurate mapping from the images to the labels, or rather until the network had a classification accuracy score of 100% on the training data. By evaluating the trained model using the classification accuracy score I was able to verify that the model had 100% accuracy when classifying the training data.

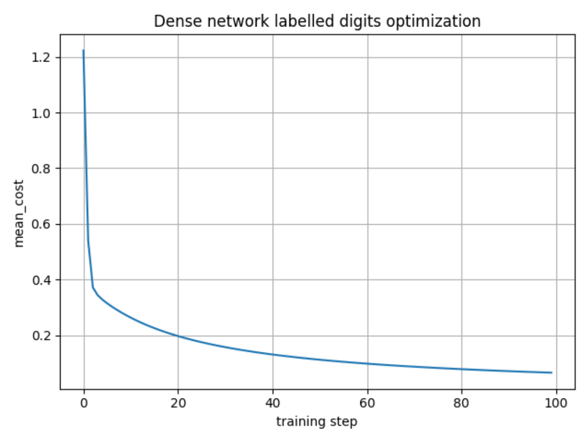


Figure 11.1. The optimization graph for a network learning to classify labelled **digits**.

For figure 11.2 I used **MNIST** partitioned into 1 part testing and 4 parts training data to test and train a dense network. The training data was shuffle-split into a number of mini-batches that matched the number of training steps. The model was then updated stochastically for each of the mini-batches.

Initially both testing and training classification accuracy scores were very low, with scores around 10%. To solve this, I changed the training routine so that the model trained for a number of epochs, meaning a number of passes over the training data. I chose to train the model for 5 epochs, shuffle-splitting the training data with every epoch.

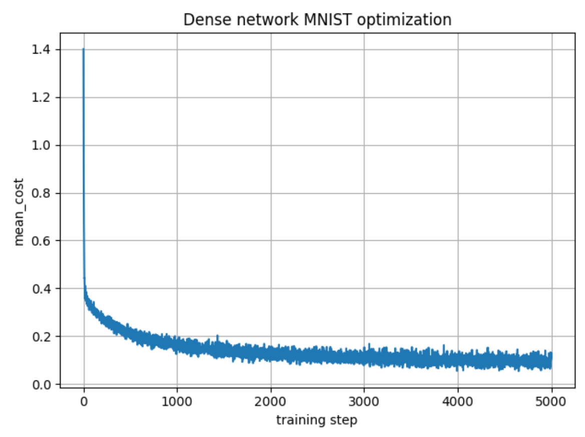


Figure 11.2. The optimization graph for a network learning to classify **MNIST**.

Test set classification-accuracy score:	**88.60%**
Train set classification-accuracy score:	**87.39%**

Figure 11.2.1. The test set and train set classification accuracy scores for a model trained for 5 epochs of 1000 steps on **MNIST** with a learning rate of 0.1.

For figure 11.3 cross-validation is used to calculate a more accurate metric for this model's performance. Cross-validation calculates the average classification accuracy score for the whole dataset by splitting it into **k** equally sized partitions, then for **k** iterations, training a version of the network with the **k**th partition as the testing data and the remaining partitions as the training data. The **k** value is referred to as the number of folds in the dataset.

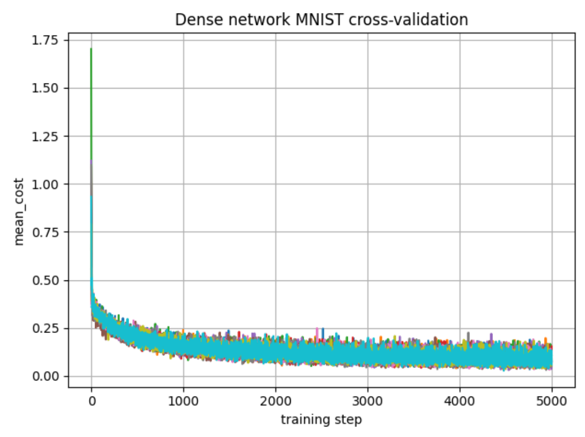


Figure 11.3. The optimization graphs for each data fold in a cross-validation of a network optimizing on **MNIST**.

Test set cross-validation accuracy score:	**87.61%**
Train set cross-validation accuracy score:	**87.88%**

Figure 11.3.1 The 10-fold cross validation accuracy scores for a model trained for 5 epochs of 1000 steps on **MNIST** with a learning rate of 0.1.

6 GAN testing

6.1 Dense GAN 2D-point generator

In the introduction I described **true_image** and **digits** as sets of multidimensional data points. To illustrate this I constructed a GAN model intended to generate a single two dimensional point and plotted the intermediate generated estimates in order to visualise the training process. Generating this two dimensional point is a similar task to generating a 10x10 dimensional point such as **true_image**, but in a way that can be represented spatially.

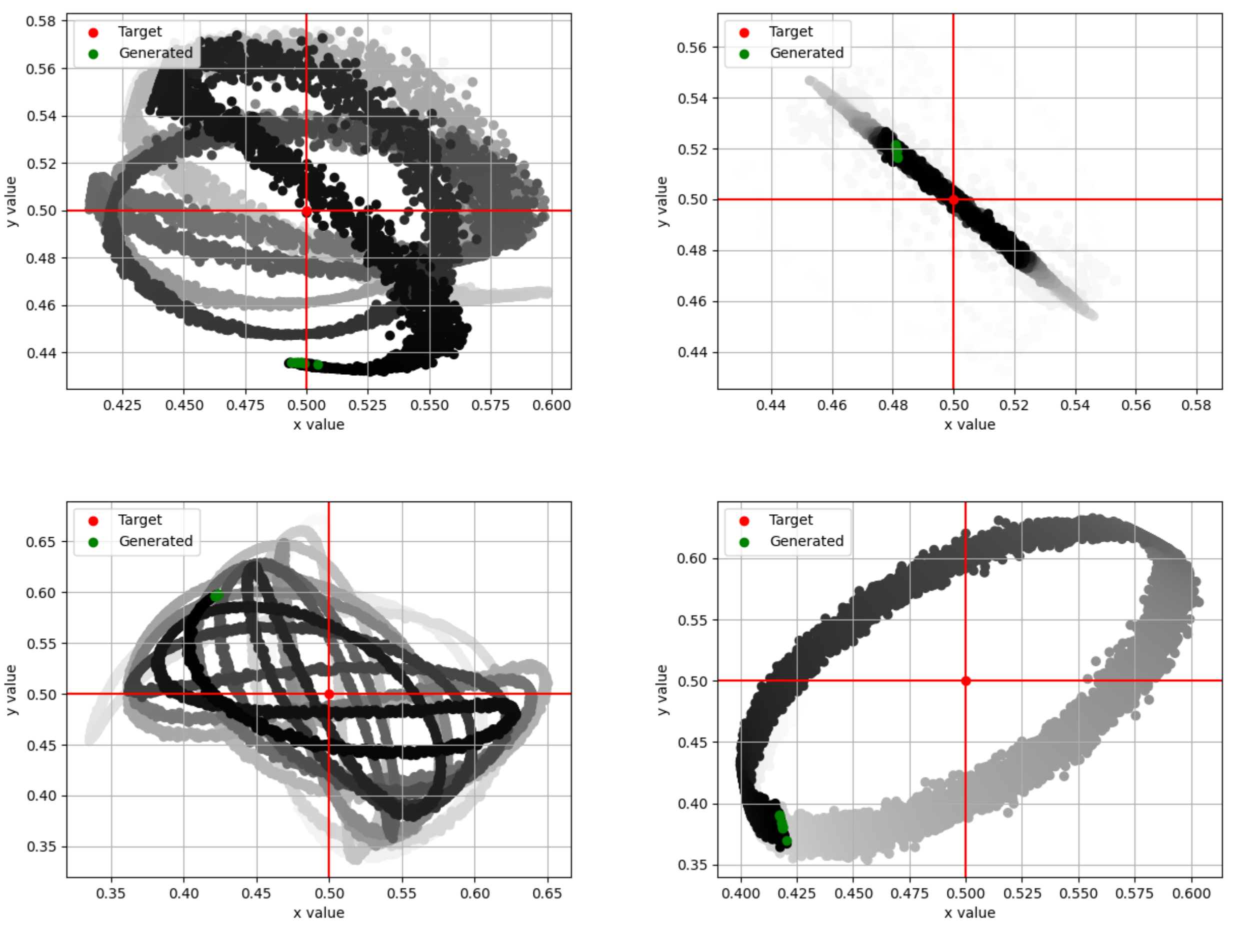


Figure 12. Two-dimensional point generator and it's attempts to capture a single data point, where points closer to white represent older estimates during training and points closer to black represent newer estimates during training. The final generated points are plotted in green and the target is marked in a red crosshair.

GANs are very difficult to balance. Illustrated in figure 12 are several different GAN configurations attempting to capture a single two-dimensional point. Only with certain configurations is the model able to capture the point precisely and even in the most precise configurations, oscillation around the correct answer is still a problem.

The models used here are all variations of a generator and discriminator both constructed of between 1-2 dense layers. (excluding the input layer) In their own context at least, these tests demonstrate that the simplest model that is still adequate for the task is usually the best model for the task.

The top right plot of figure 12 gets the closest to the correct answer and is also the simplest model of the four used here, with a {1, 2, 2} layer generator and a {2, 1} layer discriminator, trained with a learning rate of 1 with standard gradient descent. The top and bottom left plots are of a {1, 2, 2} layer generator and a {2, 2, 1} layer discriminator, both trained with a learning rate of 1. The bottom left plot is the same model, but trained with a learning rate of 0.1.

6.2 Dense GAN image generator

The first GAN model I tested was a GAN consisting of only dense layers with sigmoid activation function at each layer. Figure 13.1 shows the intermediate estimates and training graph for a GAN consisting of a {10, 200, 200, 100, 100} layer discriminator and {100, 200, 200, 200, 1} layer generator, trained with a constant learning rate of 0.05 on **true_image**.

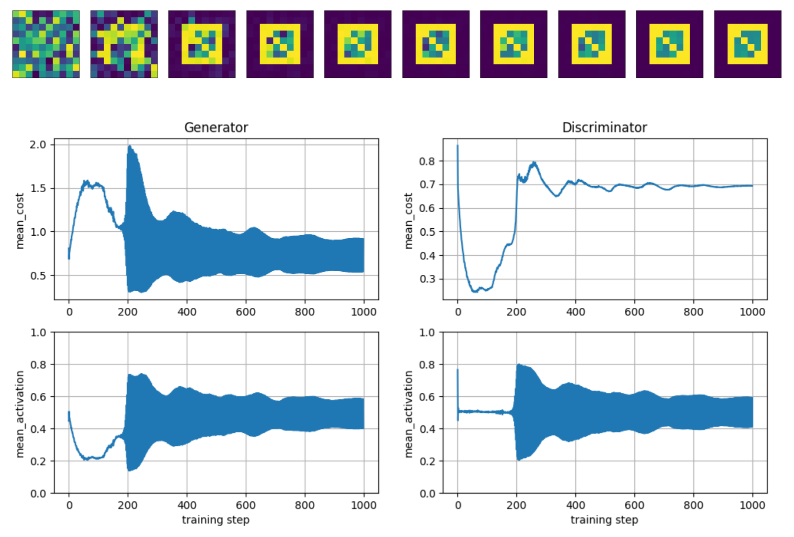


Figure 13.1. Intermediate estimates and training graph for the dense GAN model trained on **true_image** with a constant learning rate of 0.05.

In the generator column, the top plot represents the error that the generator is making in fooling the discriminator (mean generator cost) for each training step in the generator training phase. The bottom plot represents the average classification of generated samples (mean discriminator activation) for each training step in the generator training phase.

In the discriminator column, the top plot represents the error that the discriminator is making in classifying samples (mean discriminator cost) for each training step in the discriminator training phase. The bottom plot represents the average discriminator classification (mean discriminator activation) for each training step in the discriminator training phase.

In figure 13.1 at step 200, the costs and activations begin to oscillate much more than in previous steps, most likely because step 200 marks the point when the generator captures the main shapes in the image and discriminating between real and generated samples becomes a much more difficult task. From the same step in the discriminator cost plot, the cost no longer changes significantly.

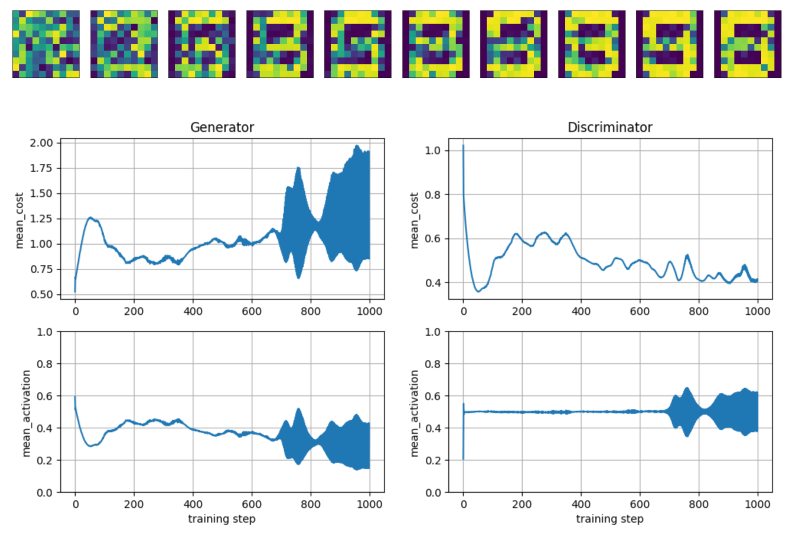


Figure 13.2. Intermediate estimates and training graph for the dense GAN model trained on **digits** with a constant learning rate of 0.05.

Figure 13.2 shows intermediate estimates and the training graph for the same model and learning rates but optimised for **digits**. During training the model captures some important features such as the regions where the digits are located in the image. The generated samples also cover a small range of modes that could very generously be called nines, eights, sixes, or threes.

The training eventually fails as a result of mode collapse, illustrated in figure 13.3. Mode collapse occurs when the generator becomes overly focused on producing samples that are similar to a limited subset of the dataset. The discriminator then learns to identify only this limited set of generated examples, and as a result, the optimization gradient it provides to the generator encourages the generator to produce even more examples that are similar to the limited set that the discriminator is familiar with. This creates a feedback loop, which leads to mode collapse.

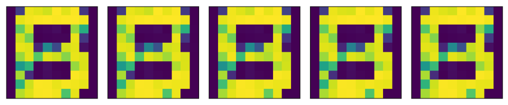
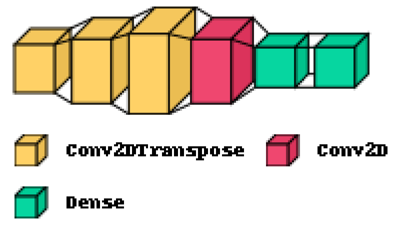


Figure 13.3. The output images from the collapsed model. The collapsed model is only able to produce a single or limited subset of images or points from the true dataset.

6.3 Convolutional GAN image generator

The second GAN model I tested was a GAN consisting of convolutional layers and dense layers with sigmoid activation functions at each layer. Plotted in figure 14.1 is a visualisation of the network. Further details of it's construction can be found in the source code, **library/test_gans.py**.



[19] Figure 14.1. Visualisation of the structure of the convolutional GAN

Figure 14.2 shows the intermediate estimates and training graph for this model, which was trained with a constant learning rate of 0.05 on **true_image**. The graph follows similar trends as in the equivalent test for the dense model. At some step, discrimination between real and generated samples suddenly becomes much more difficult as the generator finds the main shapes in the image, and the discriminator's cost no longer changes considerably.

Some visual differences between the dense model and the convolutional model include the artefacts that appear around the edges of the generated images, possibly as a result of the resolution up-sampling by the convolution operation, early in training before the network has optimized its parameters to minimize their appearance. Additionally, the convolutional model's early estimate images are less noisy than those of the dense model.

Convolutional layers process inputs based on adjacent pixel values and so the adjacent output values are smoother. The dense model does not have a notion of adjacent values so when the output is reconstructed into a 2d image the adjacent values are not smoothed.

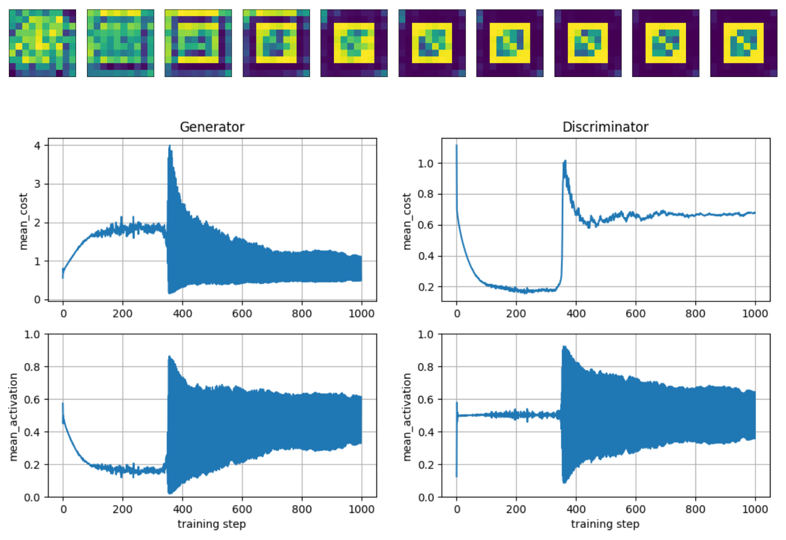


Figure 14.2. Intermediate estimates and training graph for the convolutional GAN model trained on **true_image** with learning rates 0.05

A similar difference can be seen in figure 14.3, where the intermediate estimates of **digits** are less noisy. This figure also shows the model's capture of features such as vertical and horizontal edges which could not be captured without a notion of adjacency.

For figure 14.3 I chose to show one of the faster learning rates tested (0.15) to demonstrate how in mode collapse the collapsed point usually becomes progressively higher quality. In this example the model collapsed into a point strongly resembling a six.

Before collapsing, the model manages to capture some small range of digits, including samples that could be called fives or sixes, with one sample (6th from the left) almost resembling a zero or three.

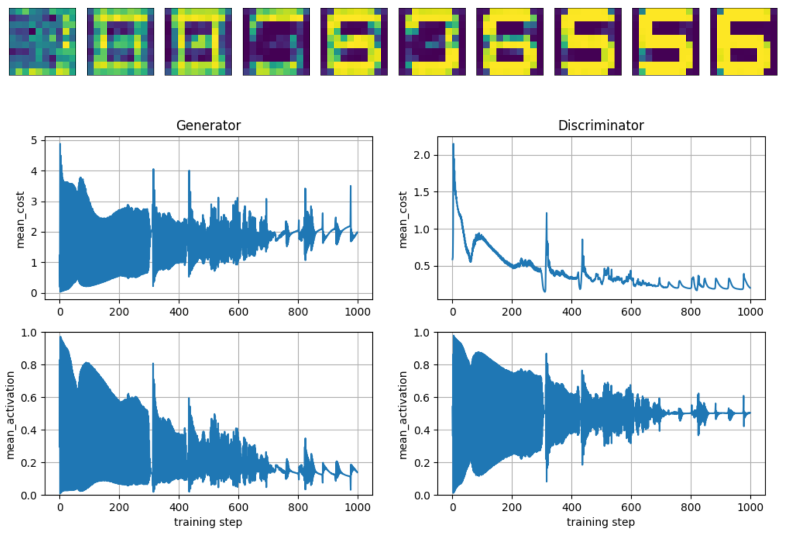


Figure 14.3. Intermediate estimates and training graph for the convolutional GAN model trained on **digits** with learning rates 0.15

In the figure 14.3 training plot, the generator cost tends upwards whilst the discriminator cost tends downwards, suggesting an overpowered discriminator. These trends show that the generator is not able to minimise cost as effectively as the discriminator, which might be one reason for the collapse of this particular training-parameter configuration.

7 Possible improvements

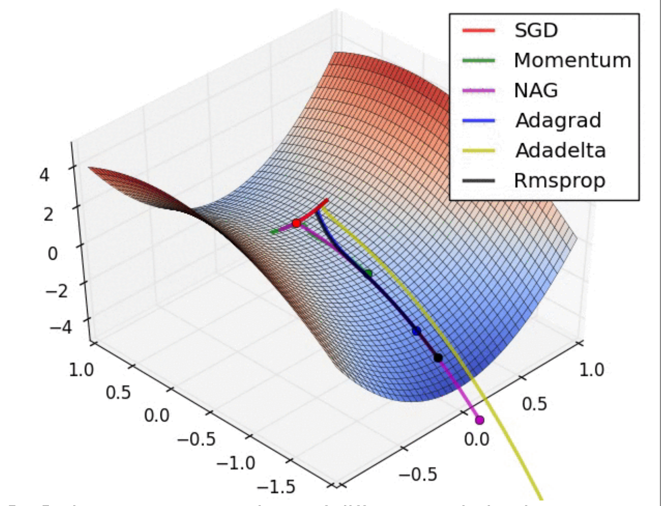
7.1 Better optimization strategy

[20] [21] Normal gradient descent can cause the network parameters to get stuck in local-minima and saddle points of the cost function topology when the slope becomes close to flat.

Gradient descent can be improved by calculating a momentum term at each step so that the behaviour of the descent path is more like a ball rolling down a hill. This allows the parameters to roll out of local minima, improving the likelihood of finding a global minima during training.

This concept is improved on by Nesterov Accelerated Gradient, which factors into the momentum term an approximation of the next gradient step, allowing the parameters to make a more predictive step towards the global minima.

Gradient descent can also be improved further by algorithms that calculate per-parameter adaptive learning rates such as Adagrad (improved on by Adadelta and RMSprop) and Adam, as well as other variations. These methods improve the likelihood of finding a global minimum by altering the parameter learning rates depending on how frequent the features modelled by those parameters are.



[22] Figure 15. Comparison of different optimization strategies on a saddle point.

7.2 GAN specific optimisation

Mode collapse is a common failure mode for GAN training. Several solutions are covered in *Improved Techniques for Training GANs - Salimans et al 2016* **[23]** which suggests a number of architectural changes that could improve the chances of convergence during GAN training. One of the changes suggested by this paper is minibatch discrimination, which describes a method of coordinating generator gradients by using a discriminator that evaluates multiple samples in combination in order to penalise the generator for generating overly-similar samples.

7.3 Better weight initialisation

The starting point of the network parameters can have a large influence on the final configuration. A good starting point is one that avoids regions of the cost topology that the optimization strategy is poorly equipped to deal with, such as saddle points and local minima. In this project, network parameters are initialised with random values sampled from the standard normal distribution, but Improved initializers such as HE and Xavier might result in better network performance. **[24]**

7.4 Regularisation and dropout

Whilst overfitting does not seem to have been a problem during this project, for more complex networks and datasets, controlling overfitting is important for finding a good general model for the target data. **[25]** Regularisation parameters add extra penalties to the cost function based on the size of the network parameters, making the model less sensitive to changes in input values and reducing variance by underfitting the model.

8 Conclusion

In this project I have set out to understand and implement an image generator as a Generative Adversarial Network. To achieve this I have adapted an external code base to better suit the project and tested neural networks written with the adapted code on simple optimization tasks and on classification tasks of both noiseless and noisy multi-modal data. I have trained different GAN models written with the adapted code on noiseless mono-modal and multi-modal image data and found that the models where all poorly equipped for finding favourable Nash equilibriums. The main improvements that could be made to the implementation are likely in the optimization strategy and in the GAN architecture.

References

1. TheIndependentCode 2021, *Neural-Network*, GitHub, viewed 10 January 2023 <https://github.com/TheIndependentCode/Neural-Network>

2. TheIndependentCode 2021, *Neural Networks*, YouTube, viewed 10 January 2023 <https://www.youtube.com/playlist?list=PLQ4osgQ7WN6PGnvt6tzLAVAEMsL3LBqpm>

3. Deng Li 2012, *The mnist database of handwritten digit images for machine learning research,* IEEE Signal Processing Magazine, vol.34, no. 6, pp. 141-142

4. Michael Neilson 2019, *Neural Networks and Deep Learning*, Michael Neilson, viewed 10 January 2023 <http://neuralnetworksanddeeplearning.com/>

5. mnielsen 2018, *neural-networks-and-deep*-learning, GitHub, viewed 10 January 2023 <https://github.com/mnielsen/neural-networks-and-deep-learning>

6. Brian McKenzie 2008, "Figure 5 Surface Plot in Matlab" in Brian McKenzie *A NEW MODEL FOR MATLAB INSTRUCTION AT OIT*, Oregon Institute of Technology, viewed 10 January 2023 <https://www.oit.edu/sites/default/files/document/a-new-model-for-matlab-instruction-at-oit---report.pdf>

7. Jason Brownlee 2019, *A Gentle Introduction to Generative Adversarial Network Loss Functions*, Machine Learning Mastery, viewed 10 January 2023 <https://machinelearningmastery.com/generative-adversarial-network-loss-functions/>

8. Google 2019, *Overview of gan structure | machine learning*, Google developers, viewed 10 January 2023 <https://developers.google.com/machine-learning/gan/gan_structure>

9. Goodfellow et al 2014, *Generative Adversarial Nets*, arXiv, viewed 10 January 2023 <arXiv:1406.2661v1 [stat.ML] 10 Jun 2014>

10. 3Blue1Brown 2018, *Neural Networks*, YouTube, viewed 10 January 2023 <https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi>

11. alexlenail 2022, *NN-SVG*, GitHub, viewed 10 January 2023 <https://github.com/alexlenail/NN-SVG>

12. Zhe Ming Chng 2022, *Using Activation Functions in Neural Networks*, Machine Learning Mastery, viewed 10 January 2023 <https://machinelearningmastery.com/using-activation-functions-in-neural-networks/>

13. Franck Dernoncourt and Stoner 2019, *What does the term saturating nonlinearities mean?*, StackExchange, viewed 10 January 2023 <https://stats.stackexchange.com/questions/174295/what-does-the-term-saturating-nonlinearities-mean>

14. Brandon Rohrer 2020, *How Convolution Works*, YouTube, viewed 10 January 2023 <https://www.youtube.com/watch?v=B-M5q51U8SM>

15. Vincent Dumoulin & Francesco Visin 2016, *A guide to convolution arithmetic for deep learning*, arXiv, viewed 10 January 2023 <arXiv:1603.07285v1 [stat.ML] 23 Mar 2016>

16. TheIndependentCode 2021, "Convolutional Neural Network from Scratch | Mathematics & Python Code" in TheIndependentCode *Neural Networks*, YouTube, viewed 10 January 2023 <https://www.youtube.com/watch?v=Lakz2MoHy6o>

17. Lei Mao 2019, *Backpropagation Through Max-Pooling Layer*, Lei Mao, viewed 10 January 2023 <https://leimao.github.io/blog/Max-Pooling-Backpropagation/>

18. Michael Neilson 2019, "Chapter 3 Improving the way neural networks learn" in Michael Neilson *Neural Networks and Deep Learning*, Michael Neilson, viewed 10 January 2023 <http://neuralnetworksanddeeplearning.com/chap3.html>

19. paulgavrikov 2021, *visualkeras*, GitHub, viewed 10 January 2023 <https://github.com/paulgavrikov/visualkeras>

20. Sebastian Ruder 2016, *An overview of gradient descent optimization algorithms*, ruder.io, viewed 10 January 2023 <https://ruder.io/optimizing-gradient-descent/index.html>

21. Diederik P. Kingma & Jimmy Lei Ba 2017, *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*, arXiv, viewed 10 January 2023 <arXiv:1412.6980v9 [cs.LG] 30 Jan 2017>

22. Sebastian Ruder 2016, “Image 6: SGD optimization on saddle point” in Sebastian Ruder *An overview of gradient descent optimization algorithms*, ruder.io, viewed 10 January 2023 <https://ruder.io/optimizing-gradient-descent/index.html>

23. Salimans et al 2016, *Improved Techniques for Training GANs*, arXiv, viewed 10 January 2023 <arXiv:1606.03498v1 [cs.LG] 10 Jun 2016>

24. Jason Brownlee 2021, *Weight Initialization for Deep Learning Neural Networks*, Machine Learning Mastery, viewed 10 January 2023 <https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/>

25. StatQuest 2018, *Regularization Part 1: Ridge (L2) Regression*, YouTube, viewed 10 January 2023 <https://www.youtube.com/watch?v=Q81RR3yKn30>