Machine Learning Manuscript

Luiz Manella Pereira

August 16, 2023

Contents

1 Introduction

This project is a lifelong project to document as much as I possibly can about machine learning as I go through learning the ins and outs of the various models. I have a GitHub (look up luizmanella on GitHub) page where I code the models using popular Python packages (and occasionally, if not too time-consuming, code them myself). The goal is not only to gain an in-depth understanding of how these models work, where they should be used, and how to code them, but to also consolidate in one place the field of machine learning so that the result is a desk manual that can be used to recall details of this broad field.

2 Optimization

2.1 Gradient descent variants

There are three variations of gradient descent, each of which uses different amount of data to compute the gradient of the objective function. The amount of data used creates a trade-off between the accuracy of the update and the time it takes to perform an update. Also, since it uses the entire dataset, it makes redundant computations when retraining the model given new data.

2.1.1 Batch gradient descent

Batch gradient descent (called vanilla gradient descent) computes the gradient of the cost function using the whole dataset, making it slow and unfeasible for datasets that are larger than the available memory. The function looks as such

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

This method also isn't usable in an *online* setting.

2.1.2 Stochastic gradient descent

Stochastic gradient descent (or SGD) instead does a parameter update for each sample in our training set. The function looks as such:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta : x^{(i)}, y^{(i)})$$

SGD is slower to converge but can be used online since it runs over individual samples. Although updates over one sample at a time causes large variations, slowly decreasing the learning rate will have the same convergence behaviour as batch gradient descent. It has been shown that SGD has better generalization at the cost of the required training time.

2.1.3 Mini-batch gradient descent

Mini-batch is in between by processing part of the data; its function looks as such:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta : x^{(i:i+n)}, y^{(i:i+n)})$$

2.1.4 General Guidelines

The following are simple guidelines when it comes to choosing the batch size for gradient descent:

- If the training set is small, simply use batch gradient descent (e.g. 2000 samples)
- For mini-batch sizes, it is good to try and choose the size as a power of 2, 2^m. Since memory works in this power of 2 structure, you can sometimes have your code run a little bit faster. It is also important to make sure the mini-batch size will be smaller than the size of your CPU or GPU memory.
- Some say using stochastic gradient descent has best generalization.

2.2 Gradient descent optimization algorithms

Out of the methods below the following is good to keep in mind. SGD with momentum seems to find more flatter minima, which tends to generalize better than sharper minima. Adaptive methods, such as Adam, quickly converges towards sharper minima.

2.2.1 Momentum

A popular method used with SGD is called **Momentum**. Rather than just using the gradient of a single step, momentum accumulates the gradient of the prior steps to determine the direction to move. It uses a coefficient of momentum that multiplies the prior gradient, which can be thought of as a percentage of the prior gradient that is retained through the next steps. Since the weight of more recent gradients is larger, we can think of this as doing an exponential average of prior gradients. This method automatically adjusts the learning rate.

2.2.2 RMSprop

RMSprop, or root mean squared propagation, like momentum, tries to dampen the oscillations. Also similar to momentum, RMSprop takes away the need to adjust the learning rate by doing so automatically. RMSprop updates based on the following steps (which are descriptions of 3 different equations):

1. The first equation computes an exponential average of the square of the gradient. Since it is done for each parameter separately, it corresponds to a component of the gradient along the direction represented by the parameter being updated.
2. The second equation is meant to decide the step size. It normalizes the values using the average of the weights.
3. The third equation is the update step.

2.2.3 Adam

Adam combines both RMSprop and Momentum, giving the best of both worlds. You compute the exponential average of the gradient and the squares of the gradient for each parameter. For the learning rate, we multiply the learning rate by the average of the gradient and divide it by the root mean square of the exponential average of square of gradients.

2.3 Initialization Techniques

Initializing the weights of a model can make a big difference during learning. How one initializes a model can affect the convergence speed, how gradients change (e.g., vanishing/exploding gradients), and overall model stability. Here we will cover some ideas regarding initialization.

- **Random sampling:** by sampling from a uniform distribution or a Gaussian distribution, we can prevent symmetrical updates across layers, which can lead to poor learning (the same reason why we do not initialize parameters to 0)

- **Xavier initialization:** this method initializes parameters in such a way that it accounts for the number of input and output neurons per layer. Given n_{in} input neurons and n_{out} output neurons, sample the weights from a Gaussian distribution with 0 mean and $\frac{2}{n_{in}+n_{out}}$ variance
- **He initialization:** this follows the same idea as above but is tailored for $ReLU$ activation and its variants. Set the variance equal to $\frac{2}{n_{in}}$
- **LeCun initialization:** this also follows the same ideas as Xavier, but is meant to be implemented with the *tanh* function. Set the variance to $\frac{1}{n_{in}}$
- **Orthogonal initialization:** When working with RNNs, setting the parameters to a randomized orthogonal matrix can help avoid vanishing/exploding gradients

3 Clustering Techniques

- PCA (Principal Component Analysis) - linear dimensionality reduction
- t-SNE (t-distributed Stochastic Neighbor Embedding) - nonlinear dimensionality reduction

4 Supervised Learning

Supervised learning is a broad topic composed of models that use labeled data to train a model. It includes both regressions and classification models. A crucial subcategory of supervised models is neural networks and it deserves a section of its own since there are so many different versions and fields that grew from it.

4.1 Linear Regression

Type of Model: Regression

A linear regression model is given by the following equation:

$$h_\theta(x) = X\theta, \quad h_\theta(x) \in \mathbb{R}^m, X \in \mathbb{R}^{m\times(n+1)}, \theta \in \mathbb{R}^{(n+1)} \tag{1}$$

The goal is to find the parameters θ such that we minimize the sum of the squared errors (also called the cost function), $SSE = \sum_{i=1}^{m}(\theta^T x_i - y_i)^2$, where x_i is the i^{th} row of the matrix X and y_i is the i^{th} target value. We can consolidate this equations as such:

$$\text{SSE} = \|X\theta - y\|^2 = (X\theta - y)^T(X\theta - y) \tag{2}$$

First, let us look at the analytical solution which is given by taking the gradient of the cost function with respect to the parameter vector, setting it to 0, and solving for the parameter vector. Since the norm is $f : \mathbb{R}^{(n+1)} \to \mathbb{R}$, we expect $\nabla \in \mathbb{R}^{1\times(n+1)}$ This is done as such:

$$\nabla_\theta \|X\theta - y\|^2 = \nabla(\theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^T y) = 2\theta^T X^T X - 2y^T X$$

Now we solve for the parameter vector as such:

$$\nabla = 0 \quad \Rightarrow \quad 2\theta^T X^T X - 2y^T X = 0 \quad \Rightarrow \quad \theta^T = y^T X(X^T X)^{-1}$$

Warning: This method requires inverting a matrix, which can be singular or computational costly. When columns are linearly dependent, the matrix cannot be inverted and thus this method cannot be used. The features of the data should be checked for high degrees of correlation prior to creating the regression. Furthermore, if the matrix is too big, inverting the matrix can take a long time and may not be the best choice to solve the regression analytically.

This brings us to the second method of solving the regression: approximating the parameters via gradient descent. Given the cost function above, we will update the parameters using the following equation until the cost converges:

$$\theta = \theta - \frac{\alpha}{m} X^T(X\theta - y)$$

Here, α is our learning rate. The choice of the learning rate needs to chosen with care. If the value is too small, it may take too many iterations until it converges. If the value is too large, the algorithm may diverge.

4.2 Logistic Regression

Type of Model: Classification

A logistic model is one that classifies data based on a sigmoidal activation function:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

The choice for activation is based on a decision boundary. Commonly, one can choose 0.5 as the boundary, where $y(x) \geq 0.5 \rightarrow y = 1$ and $y(x) < 0.5 \rightarrow y = 0$. Similar to linear regressions, we want to find the parameters θ such that our model has the best classification accuracy. To do so we will use again the gradient descent model

Here, we define the following cost function, where m is the number of training samples, or number of rows in the data set:

$$\text{Cost}(h_\theta(x), y) = -\frac{1}{m}\sum_{i=1}^{m} ylog(h_\theta(x)) - (1-y)log(1-h_\theta(x))$$

We can vectorize the equation if we consider the difference between a scalar and a vector as an element wise operation:

$$\text{Cost}(h_\theta(x), y) = -\frac{1}{m}\left[y^T log(h\theta(x)) + (1-y)^T log(1-h_\theta(x))\right]$$

The gradient descent algorithm to find the parameters can also be vectorized as such:

$$\theta = \theta - \frac{\alpha}{m} X^T \left(\frac{1}{1 + e^{-X\theta}} - y\right)$$

While this model will create a binary classification, we can extend it to a multiclass classification model. This is easily accomplished by first using the one-hot-encoding process; meaning, instead of having 1 target column with k different classes, you have k different columns, each with 1s and 0s where a 1 identifies the correct class assignment and a 0 identifies the incorrect class assignment. Then, instead of training one model, we would train a model for each target column and have k different models to make predictions on the membership of each class. When making a prediction, we would consider the highest valued prediction across our k models to be the class our new data points belong to.

4.3 Naive Bayes

Naive Bayes classifiers, although they fall under the supervised learning category, are more of a probabilistic classifier since its primary tool for classification is based on the application of Bayes' theorem. Recall that Bayes theorem states:

$$P(Y|X) = \frac{P(Y)P(X|Y)}{P(X)}$$

The primary assumption made by naive Bayes is that the features, $x_1, \ldots, x_n$, are independent in nature, which is why the model is called naive. By further expanding on the formula above, we obtain:

$$P(y|x_1, \ldots, x_n) \propto P(y)\prod_{i=1}^{n} P(x_i|y)$$

which leads to the optimal parameter which maximizes the formula just given:

$$\hat{y} = \text{argmax}_y P(y)\prod_{i=1}^{n} P(x_i|y)$$

Lastly, one can estimate $P(y)$ and $P(x_i|y)$ with maximum a posteriori (MAP) estimation.

(Interesting Details)
One of the nice things about naive Bayes is that it requires little training data and since it assumes independence, the individual distributions can be estimated without interference of any other feature's influence, which in turn combats the curse of dimensionality problem. On the flip side, while naive Bayes has been shown to be a good classifier, it has also been shown that to be a poor estimator; this mean that while the predicted classes may be accurate, the probabilities of that class being the correct classification may itself be very poor and should not be taken concretely.

5 Unsupervised Learning

Unlike supervised learning, in unsupervised learning we do not need to label the data. We can tackle problems without knowing what the results should look like by clustering the data. Clustering methods use the relationship between variables to group data points into different clusters.

6 Federated Learning

6.1 General Concepts

Federated learning (FL) is a distributed machine learning problem that generates a global model by learning from multiple decentralized edge clients. The architecture of FL is quite different than the typical machine learning problem, where data is not centrally aggregated but is distributed in a network of devices. One could assume the edge the devices on the network are themselves trying to perform learning, or they may merely be information producers while a central server is the data consumer. Regardless of the architecture, different problems present themselves, together or separately. For example, we could have edge devices with low computing power, low bandwidth, limited storage, and/or highly sensitive data (which typically you do not want to transmit). Another challenge encountered in federated learning is data heterogeneity. By data heterogeneity we mean edge devices may only ever encounter bits and pieces of the global data (which we will call the local data); this implies learning at the edge may be prone to high degrees of overfitting. The degree of unevenly distributed data across the edge devices can significantly affect the final model and is another challenge of FL. FL has various different network architectures, data-sharing architectures, and model-training architectures which are unique to different versions of the problem and require different approaches to solving them. Mathematically speaking, all of these variants have one ultimate goal, to minimize the following objective function:

$$\min_{w} F(w), \quad F(w) = \sum_{k=1}^{m} p_k F_k(w) \tag{3}$$

where m is the total number of devices, $p \geq 0$, $\sum_k p_k = 1$, and F_k is a local objective function of edge device k. Local objectives have also been called the empirical risk over the local data.

6.2 Some Solutions

There are various ways to tackle FL. First, one can consider model sharing. Model sharing simply means having a central server that passes a singular model to each agent, one at a time, to perform training before passing it on to the next. Sharing the model in the method is the most straightforward way, but not necessarily the fastest since you have to train sequentially. Another solution is called parameter averaging where each agent trains a local, but identical, model, at the same time. Once they are done, they share the parameters of their model with the trusted server which then redistributes an average of them. The averaged parameters replace the local ones creating the global model. The original paper of *FedAvg* has a few implementational details that are important. For example, rather than computing the average using all agents, they instead sample from the list of agents. More importantly, as shown in prior papers, it is key to initialize the model for each agent with the same initial parameters. Failing to do so can very likely lead to bad training with little to no chance of converging.

7 Neural Networks

7.1 General NN Topics

7.1.1 Activation Functions

Some choices for activation functions include:

- linear
- sigmoid
- Tanh
- ReLU
- Leaky ReLU
- Parameterised ReLU
- Exponential Linear Unit
- Swish - Similar to ReLU but shows significant improvement on deep architectures
- Softmax

General heuristics about choosing an activation function:

- Sigmoid functions and their combinations generally work better in the case of classifiers
- Sigmoids and tanh functions are sometimes avoided due to the vanishing gradient problem
- ReLU function is a general activation function and is used in most cases these days
- If we encounter a case of dead neurons in our networks, the leaky ReLU function is the best choice
- Always keep in mind that ReLU function should only be used in the hidden layers
- As a rule of thumb, you can begin with using ReLU function and then move over to the other activation functions in case ReLU doesn't provide with optimum results

7.1.2 Loss Functions

Some choices for loss functions include:

- quadratic
- exponential
- KL-Divergence
- cross-entropy
- hellinger distance
- Itakura-Saito

7.1.3 Neural Architecture Search

Neural architecture search (NAS) is an automatic procedure to search for neural network (NN) architectures. The goal of NAS is to search for a NN architecture that has similar, or better, performance as one designed by a human, since humans typically have search and discover the best architectures for the problem at hand. Methods for NAS can be categorized according to the search space, search strategy and performance estimation strategy used:

- The *search space* defines the type(s) of NN that can be designed and optimized
- The *search strategy* defines the approach used to explore the search space
- The *performance estimation strategy* evaluates the performance of a possible NN from its design (without constructing and training it)

7.1.4 Regularization

There are various ways to regularize your neural network, all of which are intended to improve the generalization of your network from the training to the testing data by reducing overfitting. One example is, the common, quadratic regularization:

$$\|w^l\|^2 = \sum_{i=1}^{n^{(l)}} \sum_{j=1}^{n^{(l-1)}} (w_{i,j}^{(l)})^2$$

Formulated into a neural network cost function, we have

$$J(\cdot) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i),y(i)} + \frac{\lambda}{2m} \sum_{l=1}^{L} \|w^{(l)}\|^2$$

Another form of regularization is the **dropout** regularization. The way it works is by randomly eliminating certain nodes from the network so that during training you end up training over a multitude of many smaller networks with varying architectures. There are different ways of implementing dropout, with *inverted droupout* being the most common way. You create a weight network with values from 0 to 1. You compare the values of the matrix against some probability that it will be kept (e.g. 0.8) such that the values have an 80% of being retained and a 20% of being dropped out. You then take the activated vectors, you simply multiply that outcome by this new matrix, which will reduce the size of the activation by, in this example 20%. The last step is to divide the outcome of the product mentioned by 0.8 (this number is the retain probability variable chosen by the user, and is 0.8 in our example) as to not affect the expected value of the affine transformation of the next layer. One nice intuition as to why dropout is so effective is to think about what dropping nodes randomly does with respect to feature importance. If nodes can randomly be dropped, then the neural network cannot place so much emphasis on it alone since the accuracy would be too dependent on that feature and when it is dropped, the accuracy would plummet; this would imply the neural network learns more general patterns that

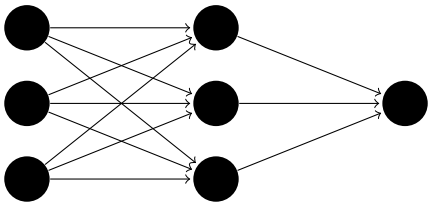
distribute this importance factor across multiple features and across the breadth and depth of the network via the salient features. Furthermore, regularization will help prevent other issues, such as vanishing gradient or gradient explosion.

Although the two methods above are the most common regularization techniques, there are other methods. An example would be **early stopping**, where you plot the training error and the test error while iterating through the training procedure, and then you choose to stop training before the test error starts to go back up (which means you've reached the point of overfitting).

7.2 Multilayer Perceptron

7.2.1 Basics

A multilayer perceptron is the most basic architecture for neural networks. A basic MLP architecture with 1 input layer (left more layer), 1 hidden layer (middle layer), and an output layer (right layer) is as such:



A neural network works as such:

1. The first layer is a vector of our features
2. We forward propagate them using activation functions. Forward propagation is simply taking the linear combination of weights (including a bias term) and the values of the layer prior and running that result through some activation function. We do this for all nodes and all layers until the output layer.
3. We then backpropagate to approximate the gradient of the loss function with respect to the weights. We need to make a choice regarding the loss function from those in the section Loss Functions. Backpropagation is simply speaking the act of computing the error associated with each layer and its contribution to the layer it feeds to and using it to approximate the gradients. These gradients are used in the gradient descent method chosen from the options listed in the Optimization section to optimize the weights.

7.2.2 Backpropagation and Gradient Checking

When implementing backpropagation, it is very possible to have subtle bugs in the code. In order to help overcome this, one can implement gradient checking. We do this by numerically approximating the derivative at each point by choosing some small ϵ (chosen sometimes to be $\epsilon = 10^{-4}$), taking the slope of the secant line between points $\theta - \epsilon, \theta + \epsilon$ as such, $\frac{d}{d\theta}J(\theta) = \frac{J(\theta+\epsilon)-J(\theta-\epsilon)}{2\epsilon}$; this method is called the two-sided difference and tends to yield better results than the one-sided method. Then we can compare this numerical derivative and compare with the results from backprop. If the results are very similar, then we know that our backprop algorithm was properly implemented and we can remove the gradient checking procedure.

7.3 Radial Basis Network

A radial basis network, or radial basis function (RBF) network, is a neural network that uses radial basis functions as its activation function. Before continuing let's first define what an RBF is. It is a real-valued function where its value only depends on the distance between input and some fixed point. Typically the distance is defined by a metric, the Euclidean metric, although it is not exclusively so. Considering the pair, RBF and metric, we now have a radial kernel which is centered around the fixed point (either the origin or some other point). The most well known RBF is the Gaussian RBF: $\exp\big[-\beta_i\|\mathbf{x}-\mathbf{c}_i\|^2\big]$

7.4 Basics

A radial basis network has a simpler architecure than most NNs. It is a 3-layer MLP (input, hidden, output) with the special activation function and an output composed of only 1 node, which is a linear combination of the the hidden units. Since the architecture of this NN is simple we show its output in one simple to understand equation:

$$\phi(\mathbf{x}) = \sum_{i=1}^{N} w_i \exp\big[-\beta_i\|\mathbf{x}-\mathbf{c}_i\|^2\big]$$

This unique architecture has been used function approximations, time series prediction, classification, and system control.

7.5 Convolutional NN

Convolutional networks, also known as convolutional neural networks, or CNNs, are a specialized kind of neural network for processing data that has a known grid-like topology. Examples include, time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals,and image data, which can be thought of as a 2-D grid of pixels. The name indicates that the network employs a mathematical operation called **convolution**, which is a specialized kind of linear operation.

Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

7.5.1 The Convolution Operation

A convolution is an operation on two functions of real-valued-arguments. If we consider an example where we measure the position of a particle through time, via function $x(t)$, we smoothe the noise of the input by averaging measurements. We make recent measurements more relevant by giving it larger weights. This operation is called a **convolution** and per our example, the convolution looks as such, where x is a function, t is time, w is a weighting function, and a is an age of measurement parameter:

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da$$

In this example, w must be a probability density function for the output to be a weighted average.

In convolutional network terminology, the first argument, x, to the convolution is often referred to as the **input**, and the second argument, w, as the **kernel**. The output is sometimes called the **feature map**.

Since continuous measurements are not necessarily always possible, we define the discrete convolution by letting the time variable be $t \in \mathbb{Z}$, which yields:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

This can be extended to multidimensional arrays, called tensors. Furthermore, we can use convolutions over more than one axis. For example, if we use a two-dimensional image I as our input, we probably also want to use a two-dimenisonal kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

Convolution is also commutative, meaning we can equivalently write

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

Usually the latter formula is more straightforward to implement in a machine learning library, because there is less variation in the range of valid values of m, n. More importantly for machine learning is the related function called the **cross-correlation**, which is the same as convolution but without flipping the kernel:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

7.5.2 Motivation

Convolution leverages three important ideas to improve an ML system:

- **sparse interactions**
- **parameter sharing**
- **equivariant representations**

Convolutional networks have **sparse interactions**, which is accomplished by making the kernel smaller than the input. This usually leads to storing fewer parameters resulting in a reduction of required memory and an improvement in statistical efficiency. It also means fewer operations to compute the output.

Parameter sharing refers to using the same parameter for more than one function in a model. Simply said, edges between 2 layers in a NN are representative of the same weight rather than independent weights.

In the case of convolution, the particular form of parameter sharing causes the layer to have a property called **equivariance** to translation. To say a function is equivariant means that if the input changes, the output changes in the same way. Mathematically speaking, a function $f(x)$ is equivariant to function g if $f(g(x)) = g(f(x))$. For convolution, if g is a function that translates the input, then the convolution is equivariant to g.

7.5.3 Pooling

A typical layer of a convolutional network consists of three stages (see Figure 1):

1. First stage are multiple convolutions to produce a set of linear activations
2. Second stage, the linear activations are run through a nonlinear activation function (e.g., the rectified linear activation function); this stage is sometimes called a **detector stage**.
3. The third stages uses a **pooling function** to modify the output of the layer further.

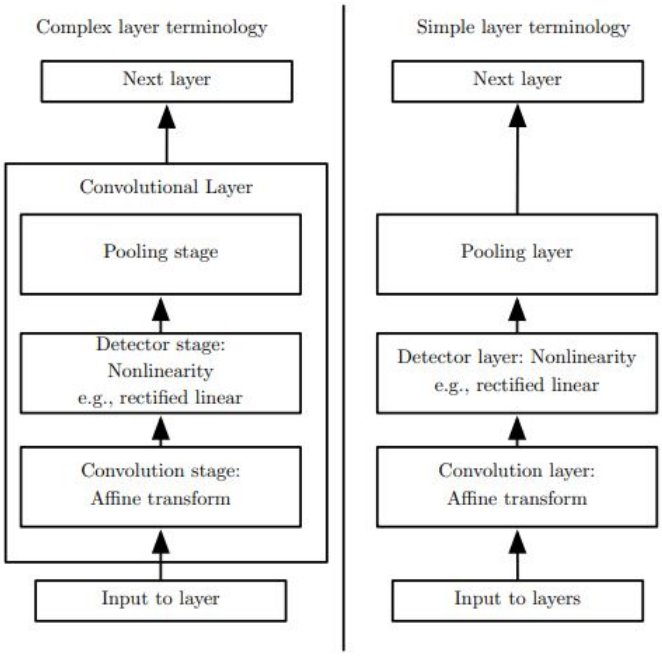


Figure 1: Three components of a typical CNN

A pooling function replaces the output with a summary statistic of the nearby outputs. For example, **max pooling** computes the maximum output within a rectangular neighborhood. Other examples include the average of a rectangular neighborhood, the L^2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel. Regardless of choice, pooling helps make the representation approximately **invariant** to small translations of the input.

Invariance to local translation can be a useful property if we care more about whether some feature is present than exactly where it is.

Pooling over spatial regions produces invariance to translation, but if we pool over the outputs of separately parametrized convolutions, the features can learn which transformations to become invariant to. If you want to know that something is exactly where it is supposed to be, the translation invariance is not to be used, although this doesn't happen extremely often.

When it comes to choosing the pooling method, some theoretical work gives guidance as to which kinds of pooling one should use in various situations (Boureau et al., 2010). It is also possible to dynamically pool features together, for example, by running a clustering algorithm on the locations of interesting features (Boureau et al., 2011). This approach yields a different set of pooling regions for each image. Another approach is to learn a single pooling structure that is then applied to all images (Jia et al., 2012).

7.5.4 Zero Padding

When performing convolutions, we shrink the data by some amount (based on kernel size). We don't always want this shrinking to happen and to avoid it we utilize zero padding. By zero padding the input we control the size output of the convolutional layer and allows us to keep using the kernel sizes of our choice. Without zero padding, the output of each layer shrinks and we end up using smaller and smaller removing our ability to choose what kernel size to use and also limiting the total number of convolutional layers in the network. There are some common choices for how much zero padding to use. These include first using no zero padding and second using enough padding to have the size of the input equal to the output (if you pass a 28x28 image, the output of the convolution is a 28x28 image).

7.6 Residual NN

Residual networks are one of the greatest solutions for truly deep networks where training the model becomes a problem due to various factors. Some of the challenges include:

- degradation problem
- vanishing/exploding gradients
- overfitting

Residual networks introduce the idea of skip connections, such as that in figure 2 Skip connections solves the

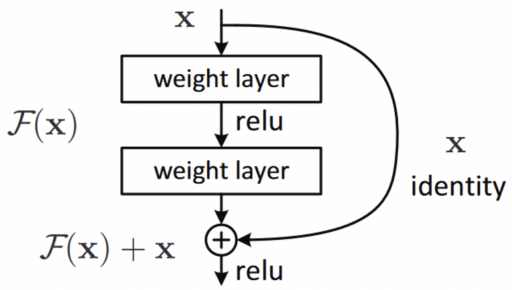
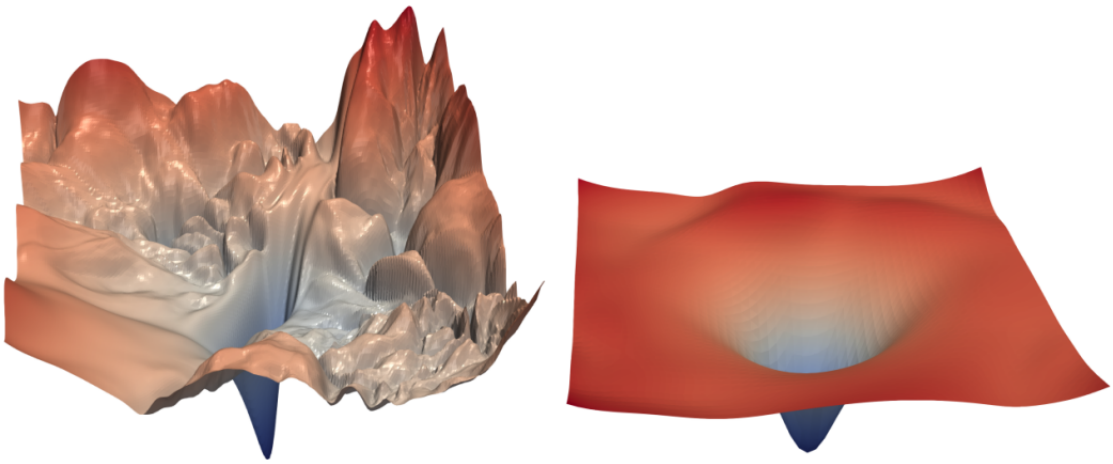


Figure 2: Skip connection example

degradation problem, where performance drops with the depth of the network architecture. Skip connections allows the model to propagate forward information from features discovered in prior steps. It has also been shown that the introduction of skip connections lead to smoother loss functions, as shown in figure 3. ResNets lead to other interesting architectures such as inception networks, and DenseNets (densely connected convolutional networks). By simply understanding residual networks, one can look further into the other models. Although they can perform better, the training time for those variations is much longer.

7.7 Mobile Net

Mobile Net is a CNN that uses a specialized convolution process to minimize the computational steps. Its goal is to perform inference on low computing environments (e.g., mobile phones). Different from traditional convolutions, Mobile Net performs a *depth-wise separable convolution* followed by a *point-wise convolution.* Depth-wise convolution means the kernel will not include all channels during this convolutional step but instead will convolve over each channel separately; if the input is 6 by 6 by 6 (channels), the kernel will be composed of 6 different matrices of dimension 3 by 3, each of which will convolve across the channels of the input respectively. The output of this step will be a 4 by 4 by 6 matrix. Next, we apply the point-wise convolution which will be a 1 by 1 by 6 tensor which will output a 4 by 4 matrix after the convolution. It is at this step that we may add additional filters to learn more latent features. Meaning, if we would like the output of the convolution to be a 4 by 4 by 10 (channels), we would have 10 different filters during the point-wise convolution. For comparison, consider an input tensor of dimension 6 by 6 by 3 and a 3 by 3 by 3 kernel with output channel, $n_c = 5$, and lastly let $f = 3$ denote the kernel dimension:



(a) without skip connections (b) with skip connections

Figure 3: Skip connection loss function

- Normal convolution would require 2160 multiplications
- Depth-wise separable convolution only requires 672 multiplications

One can find in the original paper that the ratio of the cost of the depth-wise separable convolution compared to the normal convolution turns out to be

$$\frac{1}{n_c} + \frac{1}{f^2}$$

After this architecture was created, MobileNet 2.0 was created. Its architecture includes 2 new steps. First, before performing the depth-wise separable step, it performs an expansion by applying the point-wise convolution many times to expand the number of channels to something larger. After this step it performs the depth-wise separable step and returns the number of channels back to the starting number of channels. At this point, it introduces a residual connection from input to this last output, which is the second contribution of the paper. These two additional steps still allows for inference on low-computing environments but yields better results.

7.8 Recurrent Neural Network

Recurrent neural networks, or RNN, are specialized neural networks for processing sequential data.

7.9 Autoencoders

Autoencoders (AEs) are type of neural network whose goal is to learn an encoding, or representation, or some data. It is composed of two parts, commonly called the encoder and decoder. Traditionally, the encoder will compress the data onto a smaller dimension while the decoder will decompress it back to its original state. There are many applications for autoencoders, such as dimensionality reduction and data denoising. In the GitHub repository that goes along with this we did not focus on application but rather on simply building an AE. Our design was very simple, one input, hidden, and output layer. There are other AEs that leverage deep architectures, known as Deep Autoencoders, that work very well. Regardless of depth, AEs all look similar to Figure 4.

8 Generative Models

8.1 Generative Adversarial Network (GAN)

Generative adversarial networks (GANs) is one of various different generative models but it is the model that kicked off the large spike in interest on generative models. GANs, unlike its predecessors, introduced a two models system, coined an adversarial process, where one model generates new data and the second model discriminates against its true identity (i.e., is it real or fake).

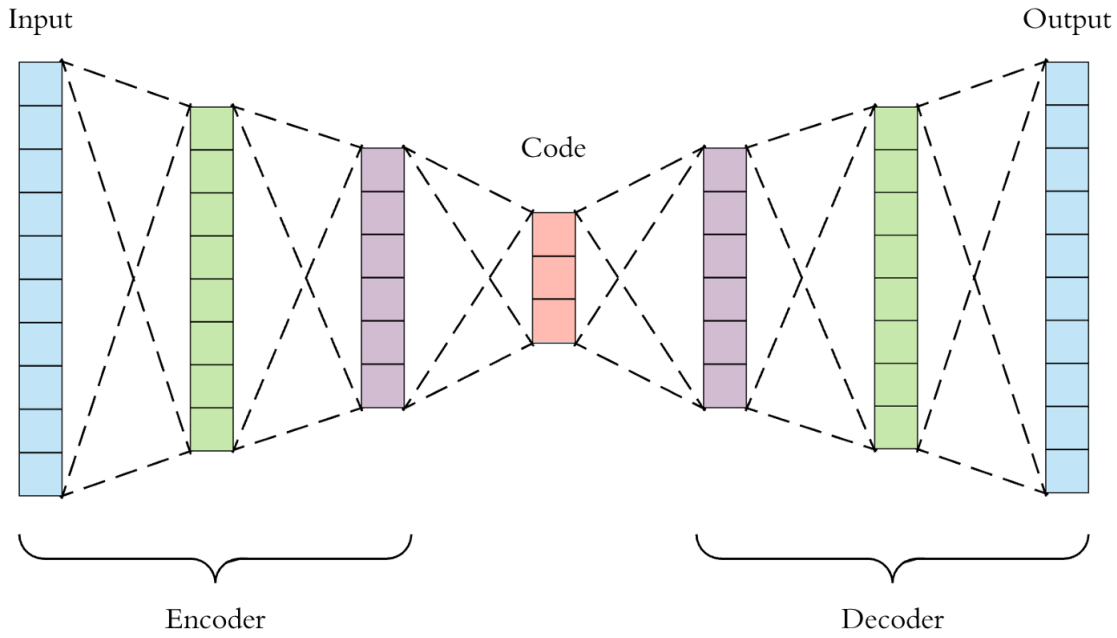


Figure 4: Autoencoder Architecture

8.1.1 Basics

The idea of adversarial modeling can be applied to different type of underlying models for both the discriminator and generator. For example, you have can Deep Convolutional GANs, but the model is more easily trained and more straightforward to understand if we consider them as plain vanilla neural networks. The goal of the model is to learn a distribution for the generator, p_g, given some dataset, $\mathbf{x}$. We begin with some noise sampled from distribution $p_z(z)$ and represent a map from this distribution to the real data as $G(z;\theta_g)$. Note that $G(\cdot)$ must be a differentiable function, thus using MLPs we can utilize backpropagation to approximate its differential. Next, we define the discriminator, which is also an MLP, as $D(x;\theta_d)$. The discriminator outputs a real number which represents the probability that $\mathbf{x}$ is a sample from the real dataset and not a synthetic sample generated by the generator. The goal during training is to maximize the discriminators ability to correctly label the fake and real points while minimizing $\log(1 - D(G(z))$. The game played between the discriminator and the generator is called a two-player minimax game defined as follows:

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

8.1.2 Comments

There are a few interesting things about the model that one can find out while reading various research papers and watching lectures on this topic. One is that the accuracy of the discriminator converges to 50%. The reason for this is that the generator is trying to match as closely as possible its distribution to that of the data. As the generator improves, the synthetic data it generates becomes increasingly closer and closer to the real data and thus the discriminator will essentially be guessing whether a new sample passed to it is real or fake. Another interesting point to mention is how to choose the initial distribution to generate noise from. If one looks up lectures on this topic from Ian Goodfellow he explains that the choice is virtually irrelevant and that choosing the simplest possible distribution, a uniform distribution, yields as good a result as any other choice; hence why most examples use the uniform distribution for sampling noise.

8.2 Wasserstein Generative Adversarial Network (WGAN)

Wasserstein Generative Adversarial Networks, or WGANs, build on the theory provided by GANs. While both achieve the same goal, to create a generative model, WGANs focus instead on how to measure the distance between the model distribution and the real distribution; this metric is important when we speak about the convergence of a distribution onto another distribution. The distance that is considered is the Earth-Movers (EM) distance, also known as the Wasserstein-1:

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

where $\Pi(\mathbb{P}_r, \mathbb{P}_g)$ is the set of all joint distributions with marginals $\mathbb{P}_r$ and $\mathbb{P}_g$. In optimal transport theory we think of this distance as moving some amount of mass from one place, x, to another, y, where the cost to transport is minimal. As described in the paper that introduced WGANs, "Wasserstein-GAN minimizes a reasonable and efficient approximation of the EM distance." As we will see, by changing the loss function for the discriminator we will essentially solve the Wasserstein problem.

8.2.1 Basics

If one dives deeper into Optimal Transport theory, they would run into the question of whether the Wasserstein distance between two distributions, $W(\mathbb{P}_r, \mathbb{P}_\theta)$, is computable. The problem is indeed intractable and so, like many other papers, the authors turned to an alternative, dual, formulation:
(*Kantorovich-Rubinstein Duality*)

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

where the supremum is over 1-Lipschitz functions, $f : X\mathbb{R}$, and

$$\nabla_\theta W(\mathbb{P}_r, \mathbb{P}_\theta) = -\mathbb{E}_{z \sim p(z)}\big[\nabla_\theta f(g_\theta(z))\big]$$

The function above dictates the loss function for the discriminator. The final piece to complete the WGAN model is to restrict the sets of functions we consider to be solutions of the Wasserstein-1 distance by adding a constraint such that after each gradient update, the weight parameters of the discriminator, also called the critic, must fall within a hypercube (e.g., $W = [-0.01, 0.01]^l$).

8.2.2 Comments

WGAN, unlike GAN, requires training the critic, or discriminator, 5 times more than the generator. So, for every loop during the training of the WGAN, the critic receives 5 times more updates than the generator.

9 SVM

Support vector machine: a classification or regression model that make classifications or predictions by utilizing a hyperplane that separates our data based on support vectors. Simply put, given the dataset $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$,

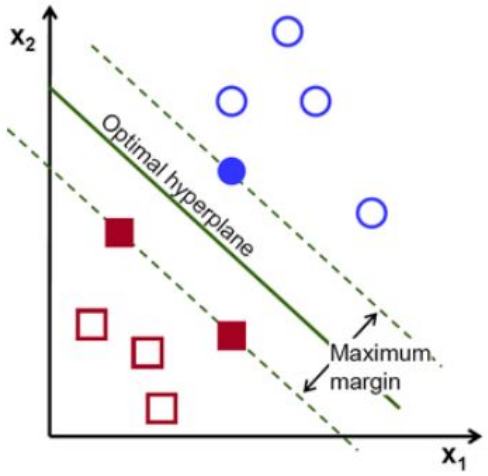


Figure 5: Basic idea of SVM model.

we look for a solution for the following equation (that of a hyperplane):

$$\mathbf{w}^T\mathbf{x} - b = 0$$

where $\mathbf{w}$ is a normal vector to the hyperplane. If the data is linearly separable, then one can say the solution contains what we call a hard margin; should the data not be linearly separable, then we are dealing with a soft margin problem where we utilize the hinge loss function and focus on minimizing the following optimization problem:

$$\lambda\|\mathbf{w}\|^2 + \left[\frac{1}{n}\sum_{i=1}^{n} \max\big(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i - b)\big)\right]$$

Lastly, when working with SVMs, it is sometimes better to apply a kernel trick in order to make a dataset linearly separable. Figure 6 illustrates how a kernel trick can turn non-linear data linear. There are various standard kernels one can use. The following are some:

1. Polynomial kernel
2. Gaussian kernel
3. Sigmoid function or hyperbolic tangent

Although there are other kernels, and specialized kernels built for specific problems, they seldom appear in common research and application simply due to the difficulty of finding ones that work.

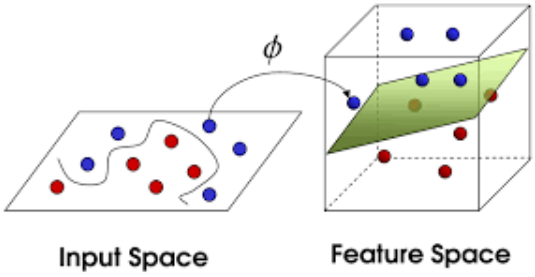


Figure 6: Example where using kernel trick linearizes data

10 Appendix

10.1 Evaluating Learning Algorithms

Evaluation can give a lot of information about a model and can help identify problems. For example, if a model does not predict well on new data, there are several options available to improve this model (e.g., more training data, more features, less features). Evaluation methods helps to make an informed decision on how to fix the model.

The standard way to evaluate a model is by splitting your training data (i.e., 70/30), training over the new smaller training data, then computing the error on the test data to see the accuracy of your model. If possible, it is a good idea to shuffle the data around prior to doing the split.

11 List Of Things Related to ML to research

Types of learning problems

- Hybrid Learning
  - semi-supervised learning
  - self-supervised learning
  - multi-instance learning
- Statistical Inference
  - inductive learning
  - deductive inference
  - transductive learning
- Learning Techniques
  - multi-task learning
  - active learning
  - online learning
  - transfer learning

  - ensemble learning

The following is a list of topics I plan on covering:

- Siamese networks
- variational recurrent neural network
- deep recurrent neural network
- long/short term memory (LSTM)
- gated recurrent unit (GRU)
- variational AE (VAE)
- denoising AE (DAE)
- sparse AE (SAE)
- markov chain (MC)
- hopfield network (HN)
- boltzmann machine (BM)
- restricted bm (RBM)
- deconvolutional network (DN)
- deep convolutional inverse graphics network (DCIGN)
- liquid state machine (LSM)
- deep residual network (DRN)
- Kohonen network (KN)
- neural turing machine (NTM)
- extended physics informed neural networks (EPINN)
- Attention models and Transformers

Other things I want to look at:

- Neural Architecture Search
- evolutionary algorithms
- hyperparameter tuning
- feature selection
- feature engineering
- swarm intelligence

References