Notes on Machine Learning and Neural Networks

Shailesh Lal

shailesh.hri@gmail.com

Abstract

These notes are an introduction to Machine Learning biased towards Neural Networks, and summarize a short lecture course given to Graduate and Masters students in theoretical physics. We are motivated by the many recent applications of these methods to several key problems in theoretical and mathematical physics, but the lectures themselves stick to introducing the theory and application of these methods at a level of rigour accessible to a Masters student.

Contents

1 References **3**

2 Learning from Data **3**

3 Regression *via* Generalized Linear Models **7**
3.1 Linear Regression . . . 8
3.2 Bias and Variance . . . 9
3.3 Regularization . . . 12

4 Optimization **14**
4.1 Gradient Descent . . . 15
4.1.1 Stochastic Gradient Descent . . . 16
4.2 Gradient Flows . . . 16

5 Classification *via* Logistic Regression **17**
5.1 Binary Classification . . . 17
5.2 Multinary Classification . . . 19

6 Neural Networks **21**
6.1 The Perceptron . . . 21
6.2 Artificial Neural Networks . . . 25
6.3 How Neural Networks Learn: Backpropagation . . . 27
6.4 Vectorized Backpropagation . . . 30

7 Training a Neural Network **30**
7.1 The Vanishing Gradients Problem . . . 30
7.1.1 Initialization . . . 31
7.1.2 Non-Saturating Activation Functions . . . 34
7.2 Zero Centering Activations . . . 34
7.3 Batch Normalization . . . 36
7.4 Choosing the Learning Rate . . . 37

8 Convolutional Neural Networks **39**
8.1 A Simple Edge Detection Problem . . . 41
8.2 Convolutional Layers . . . 42
8.3 Pooling Layers . . . 46
8.4 Putting it all together: A basic ConvNet . . . 47
8.5 A Network within a Network . . . 48
8.6 Convolutions are All You Need . . . 49

9 Conclusions & Outlook **50**

A Bias/Variance Decomposition **50**

B A Bayesian Treatment of Regression 53
B.1 Maximum Likelihood Estimation 54
B.2 Maximum Posterior (MAP) 55
B.3 A Fully Bayesian Treatment 55

C Entropy and Cross-Entropy from Information Theory 57

1 References

These notes are intended to be an overview of some machine learning methods biased towards neural networks. As such, they draw on several references which would be useful for further reading. In particular, we would like to suggest [1] and [2] for practical implementations in Python along with [3], [4] and [5] for the more formal aspects. Further, a comprehensive introduction to machine learning for physicists is [6]. Finally, there are several reviews for the applications of machine learning to mathematics and mathematical physics broadly defined [7, 8, 9, 10, 11]. We also refer the reader to the very engaging [12] in which many initial computations involving machine learning and Algebraic Geometry and String Theory were performed concurrently with [13, 14].

2 Learning from Data

The ability to learn from data is the cornerstone of modern science. The list of experiments that played a direct role in the formulation of deep physical principles and theories is wide, spanning among others Brahe's observations of planetary motion (gravity), Millikan's oil drop experiment (quantization of electric charge), the photoelectric effect (wave-particle duality) and atomic spectra of Hydrogen (the Bohr model). What is perhaps less appreciated is that deep theoretical insight may also follow by analyzing physical phenomena as a collective whole, rather than individual instances.

Perhaps a prototypical example of this approach is the periodic table constructed by Mendeleev by organizing all the known elements at the time into what are essentially equivalence classes, the columns of the periodic table. Armed with this table, not only was it possible to predict the properties of elements from those of their neighbours, one could also predict the existence of *new*, i.e. previously unobserved, elements which were indeed found subsequently. Even though the periodic table proposed by Mendeleev was later supplanted by the modern periodic table which classifies elements not by atomic weight but by atomic number, the ideas proposed by Mendeleev sit at the foundation of modern physics and chemistry [1]. This remarkable triumph of reasoning also led to the observation of the octet rule and full-filled and half-filled stability which are key to the quantum mechanical description of atoms.

In more modern times, string theory in particular and mathematical physics in general sit on the cusp of a data revolution. Particularly, in algebraic geometry there has been a long and sustained numerical effort dating back to the 1980s to exhaustively compute and classify the topological properties of Calabi Yau Manifolds. Geometric properties, though far more prohibitive, are also now within reach. This research program clearly has a signficant interplay with string phenomenology where the construction of Calabi Yau manifolds with desired topology and geometry is clearly of great importance for the search for the Standard Model in the string landscape.

[1]These efforts to bring organization to modern Physics and Chemistry may, only slightly fancifully, be regarded as a successful resolution of the first landscape problem encountered in science.

It is perhaps not surprising then that the *espirit de temps* of machine learning has also lent itself to our efforts in mathematics and theoretical physics. This is primarily for two reasons. Firstly, as our computational power increases, so does the the amount of data that we are able to generate and need to analyze. This comes somewhat fortuitiously at a time when the automated analysis of data using machine learning is a rapidly growing field. Secondly, machine learning methods provide us a way to accurately estimate the values of certain target variables in real time as opposed to traditional approaches for computing the exact answer which can be double exponential complexity or even NP hard. In contrast, machine learning can often train to learn an approximate answer in polynomial time. Clearly this would be of great interest in speeding up searches for the Standard Model within the string landscape.

These lectures are largely agnostic on the possible applications and we have chosen, somewhat deliberately, to present instead stand-alone methods from modern machine learning with a focus on the general principles and assumptions that underly these methods along with more practical details of their implementation. However, the methods themselves have been chosen as they are among the most successfully applied frameworks to study problems in the string landscape and more generally, mathematical physics. We will in particular focus on the problem of *supervised machine learning*, where our data comprises of inputs x represented as vectors in $\mathbb{R}^D$,

$$x^T = (x_1, x_2, \ldots, x_D) \ , \tag{2.1}$$

each of which have an output t corresponding to it. This output may be a continuous variable, i.e. a *regression* problem, or a discrete label which corresponding to the corresponding datum, i.e. a *classification* problem. In either case, t may also be a K-tuple valued in $\mathbb{R}^K$. We will assume that we have a finite amount of data $\mathcal{D}$ consisting of N input-output pairs, i.e.

$$\mathcal{D} = \{(x_\alpha, t_\alpha)\} \ . \tag{2.2}$$

We would like to develop a framework for predicting the value of t given the value of x and the data $\mathcal{D}$. In the following we will focus on some general features of this framework.

The first element of our proposed framework is that in lieu of developing explicit algorithms to compute t given x we will aim to *learn by experience*, i.e. to utilize our knowledge of N previously known input output pairs (x_α, t_α) to make new predictions. We assume that there is some underlying regularity that governs the outputs t, but also that there may be random fluctuations in t appearing as noise. We may model this scenario by postulating the existence of an unknown joint probability distribution $p\,(x, t)$ from which the pairs (x_α, t_α) are independently drawn. Then, for a fixed value of x, t is not deterministic but is instead a random variable characterized by the conditional probability distribution $p\,(t|x)$. In practice of course the joint probability distribution function is almost never known to us and all that we have is the data $\mathcal{D}$ of (2.2) to learn to predict with.

In order to do so, one consider prediction functions $f\,(x)$ which output candidate values for t given a value x. We also define $\mathcal{F}$, the set of all prediction functions [2]

$$\mathcal{F} = \{f_\theta \, , \, \theta \in \Theta\} \ , \tag{2.3}$$

indexed by the set Θ. Next, in order to characterize the performance of a given predictor f, we define a *loss* which measures the discrepancy between the true output and the prediction. There is

[2]In practice we restrict $\mathcal{F}$ to a *hypothesis class* such as polynomials of a given degree, in which case the θ are polynomial coefficients. Ultimately it is also possible, and desirable, to compare across multiple hypothesis classes.

no canonical choice for such a loss function, though some choices are *de rigeur*. For example, given an input x, an output t and a prediction $f(x)$ for a function f drawn from $\mathcal{F}$, one may define

$$L(t, f(x)) = \begin{cases} 1\,, & f(x) \neq t\,, \\ 0\,, & f(x) = t\,. \end{cases} \tag{2.4}$$

One may equally well define instead

$$L(t, f(x)) = (t - f(x))^2\ . \tag{2.5}$$

The optimal prediction function, denoted $\tilde{f}$, is the one which minimizes the expected loss. i.e.

$$\tilde{f} = \underset{f \in \mathcal{F}}{\operatorname{argmin}}\, \mathrm{E}\left[L(t, f(x))\right]\ , \qquad \mathrm{E}\left[L(t, f(x))\right] = \int L(t, f(x))\, \mathrm{d}p(x,t)\ . \tag{2.6}$$

In practice, such expectation values can hardly ever be computed as the joint probability distribution function $p(x,t)$ is unavailable to us. Instead, we make an empirical estimate of the expected loss by computing an expectation value of the loss over the given data $\mathcal{D}$ in (2.2), i.e.

$$\mathrm{E}_{\mathcal{D}}\left[L; f\right] = \frac{1}{N} \sum_{\alpha=1}^{N} L(t_\alpha, f(x_\alpha))\ , \tag{2.7}$$

and seek to determine $\tilde{f}$ by minimizing $\mathrm{E}_{\mathcal{D}}$, i.e. we propose

$$\tilde{f} \approx \underset{f \in \mathcal{F}}{\operatorname{argmin}}\, \mathrm{E}_{\mathcal{D}}\left[L; f\right]\ . \tag{2.8}$$

While in principle this is perfectly well defined, in practice this presciption is plagued with the problem of *overfitting*. This is because it is possible for ansatz functions f to simply *memorize* the available data. In other words, it might be the case that there exists a choice of $\hat{\theta}$ for which

$$t_\alpha = f_{\hat{\theta}}(x_\alpha)\ , \tag{2.9}$$

on a case by case basis over the entire dataset $\mathcal{D}$ [3] Our prescription would pick out $f_{\hat{\theta}}$ as the candidate for $\tilde{f}$ even though predictions $f_{\hat{\theta}}(x)$ might otherwise be far from the actual values of t. The true expected loss when predicting with $f_{\hat{\theta}}$ would then be very large and far from the minimum. To get around this problem, we need an estimate for how well our candidate for $\tilde{f}$ generalizes to the full dataset. This, by definition, involves estimating the performance of a given predictive function

[3]For intuition, consider the case of learning data $\mathcal{D} = \{(x_\alpha, t_\alpha)\}$ where x and t are scalar variables. We also restrict $\mathcal{F}$ to a hypothesis class of polynomials of degree M, in which case

$$f_\theta = \sum_{k=0}^{M} \theta_k\, x^k\,. \tag{2.10}$$

Intuitively, if M is large enough it might be possible that there exists an M-tuplet $\hat{\theta}$ such that

$$t_\alpha = f_{\hat{\theta}}(x_\alpha) \qquad \forall \qquad (x_\alpha, t_\alpha) = \mathcal{D}\,. \tag{2.11}$$

There is no guarantee however that this good performance has been obtained by truly learning the underlying regularities of data or by having a large enough number of tunable model parameters which allow us to reproduce known data on an individual basis. See Figure 3 for an explicit example, and Section 3 below for more details.

over unknown data. A common strategy for making this estimate is to perform a *train-test split*, i.e. split the available data into a training subset $\mathcal{D}_{train}$ and a test subset $\mathcal{D}_{test}$, typically by random partitioning. We then compute

$$\tilde{f} \approx \hat{f} = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \, \mathrm{E}_{\mathcal{D}_{train}} \left[L; f \right] . \tag{2.12}$$

Having determined a candidate $\hat{f}$ we can compute both $\mathrm{E}_{\mathcal{D}_{train}} \left[L; \hat{f} \right]$ and $\mathrm{E}_{\mathcal{D}_{test}} \left[L; \hat{f} \right]$ as per (2.7). The $\mathrm{E}_{\mathcal{D}_{test}}$ is then our estimate of the performance of our predictor on unknown data. As a criterion for an acceptable solution, we may require for instance that

$$\mathrm{E}_{\mathcal{D}_{train}} \left[L; \hat{f} \right] \approx \mathrm{E}_{\mathcal{D}_{test}} \left[L; \hat{f} \right] , \tag{2.13}$$

i.e. the function $\hat{f}$, which is our candidate for $\tilde{f}$, performs comparably on the data $\mathcal{D}_{test}$. If so, one may be reasonably confident that our predictions generalize well to unknown data. This approach is shown as a schematic in Figure 1. *K-fold cross-validation* is a more sophisticated extension of this approach, which we now outline.

1. partition $\mathcal{D}$ into k disjoint subsets, or k *folds*, $\mathcal{D}^{(k)}_{test}$.
2. This defines k training sets $\mathcal{D}^{(k)}_{train}$ by taking the complement of $\mathcal{D}^{(k)}_{test}$ in $\mathcal{D}$, i.e.
$$\mathcal{D}^{(k)}_{train} = \left\{ d \in \mathcal{D} \,|\, \quad d \notin \mathcal{D}^{(k)}_{test} \right\} . \tag{2.14}$$
3. Make k estimates for $\hat{f}$, denoted by $\hat{f}^{(k)}$, by computing
$$\hat{f}^{(k)} = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \, \mathrm{E}_{\mathcal{D}^{(k)}_{train}} \left[L; f \right] . \tag{2.15}$$
4. Next, compute the corresponding test loss $\mathrm{E}_{\mathcal{D}^{(k)}_{test}} \left[L; f^{(k)} \right]$.
5. We therefore have k estimates of the model performance in the wild, which should give us a better idea of the generalizability of the model than a single number.

Theoretically this approach presents a challenge because data is being reused in $\mathcal{D}^{(k)}_{test}$ and thus the estimates of the expected loss are not independent of each other. Practically, this is a popular framework especially when estimates $\hat{f}$ can be arrived at in reasonable computational time. So far we have outlined the general approach of supervised learning to address prediction problems in datasets. However, our discussion has been accordingly heuristic and focused on general principles. We will now turn to the study of some machine learning frameworks with which we can concretely implement the above approach. While our ultimate interest is in neural networks, we will start with conceptually simpler models both as a warm up and because they exhibit properties that directly feed into the construction of neural networks.

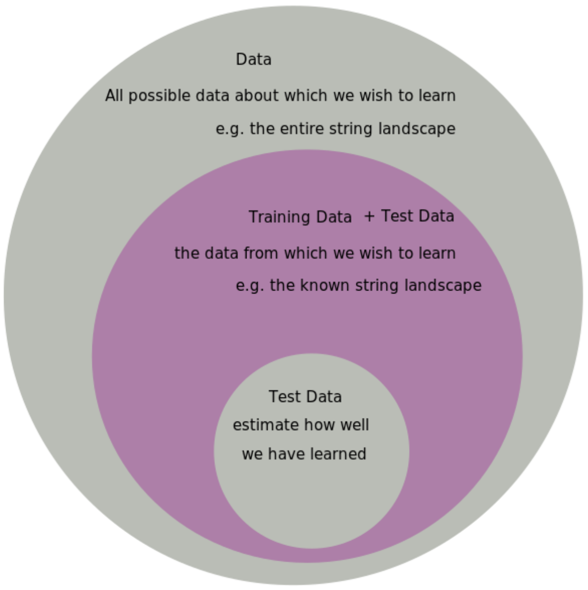

Data
All possible data about which we wish to learn
e.g. the entire string landscape
Training Data + Test Data
the data from which we wish to learn
e.g. the known string landscape
Test Data
estimate how well
we have learned

Figure 1: Learning from Data. The outer gray circle represents the entirety of data that could in principle be available to us, and about which we wish to learn. Learning involves studying available data, the inner pink circle, to elucidate patterns and correlations on the basis of which we can make predictions about all data. The inner-most circle represents a subset of available data that we keep aside for testing our predictions on.

3 Regression *via* Generalized Linear Models

In this section we will define and study the so-called generalized Linear Models for regression as precursors to neural networks. Doing so allows us to explicate many general issues that arise in using *any* machine learning framework to learn from data at hand. Our data comprises of inputs x represented as vectors in $\mathbb{R}^D$,

$$x^T = (x_1, x_2, \ldots, x_D) \ , \tag{3.1}$$

each of which have an output t corresponding to it. Since we have a regression problem we will assume that t is a continuous variable. As stated at the outset, we would like to develop a framework for predicting the value of t given the value of x by utilizing our knowledge of N previously known input output pairs $\mathcal{D} = \{(x_i, t_i)\}$. We assume that the outputs t arise from an underlying regularity with random fluctuations, i.e.

$$t = y\,(x) + \epsilon\,, \tag{3.2}$$

where $y\,(x)$ is the underlying regularity we seek to model. and ϵ is a random noise. We will seek to model $y\,(x)$ by selecting from a family $y\,(x, w)$ of functions parametrized by a tuplet w using the principles outlined in Section 2.

At the outset, it should already be apparent that the quality of predictions we can make for t depend on both the ansatz that we choose for $y\,(x, w)$ and the value of the parameters w for the given ansatz. In this section we will describe some guiding principles for choosing the ansatz y and the values w in order to model the underlying regularity in data as well as possible. As mentioned above, these will be very general principles that present themselves in far greater complexity when working with neural networks.

3.1 Linear Regression

Perhaps the first machine learning problem ever studied was the following: given some observations

$$\mathcal{D}_{train} = \{(x_\alpha, t_\alpha)\} \qquad \alpha = 1, 2, \ldots, N\,, \tag{3.3}$$

estimate $t(x)$ by fitting a straight line

$$y = w_0 + w_1\, x\,, \tag{3.4}$$

through $\mathcal{D}_{train}$ where w_0 and w_1 are the slope and intercept of the line $y\,(x)$. These parameters are fixed by requiring that the mean squared distance between the observations t_n and the predictions $y\,(x_n)$ are minimized. That is, we define the *cost function*

$$E\,(w) = \frac{1}{2}\sum_{\alpha=1}^{N}\left(y\,(x_\alpha; w) - t_\alpha\right)^2\,, \tag{3.5}$$

and then (w_0, w_1) is the solution of

$$\nabla_w\, E\,(w) = 0\,. \tag{3.6}$$

We may explicitly solve for the minima w_* by solving (3.6) in w. The solution is given later in (4.5) for a more general problem. Hence a candidate function $y\,(x; w_*)$ can always be determined, and predictions for t can then be made *via*

$$t\,(x) = y\,(x, w_*)\,. \tag{3.7}$$

Having done so, a number of questions naturally arise. Firstly, choosing the ansatz (3.4) means we can only fit straight lines through the data. But the data may be more complex, and t and x may have a complicated non-linear relation which this ansatz is unable to capture. Secondly, the overall goal of machine learning is to make good predictions for *new* data, rather than doing arbitrarily well on the known data. How do we estimate how well our model will do in the wild? Thirdly, if we can indeed extend this framework to infer complicated relationships between x and t then how do we prevent the model from spurious features in our training data that may not be present in the overall dataset? These questions are deeply entwined with several key concepts in machine learning, as already discussed in Section 2.

To begin with, we point out that the ansatz (3.4) can be easily extended to polynomial functions. In particular, we may take $y\,(x; w)$ to be a degree M polynomial

$$y\,(x; w) = \sum_{j=0}^{M} w_j x^j\,. \tag{3.8}$$

Still more generally, one may choose j fixed nonlinear functions $\phi_j\,(x)$ and take the ansatz

$$y\,(x; w) = \sum_{j=0}^{M} w_j \phi_j\,(x)\,, \qquad \phi_0\,(x) \equiv 1\,. \tag{3.9}$$

While the ansatze (3.8) and (3.9) are capable of expressing rich nonlinear dependencies between x and t, they are still linear in the parameters w, and the cost function (3.5) is still quadratic in w.

Hence they may be optimed in much the same way as (3.4). Ansatze of the form (3.9), which clearly contain (3.8) and (3.4) as special cases, are collectively known as Generalized Linear Models.

The generalization to the case when x are vectors drawn from $\mathbb{R}^D$ is also apparent. We again define M scalar functions $\phi_j(x)$, now of D variables $x_1, \ldots, x_D$ and choose the same ansatz (3.9) as before i.e.

$$y(x; w) = w \cdot \phi(x) \; , \qquad w^T = (w_0, w_1, \ldots, w_M) \; , \qquad \phi^T = (\phi_0, \phi_1, \ldots, \phi_M) \; . \tag{3.10}$$

The case of ordinary D-variate linear regression is covered by the ansatz

$$y(x, w) = w_0 + \sum_{i=1}^{D} w_i \, x_i \, , \tag{3.11}$$

i.e. $M = D$ and $\phi_j(x) = x_j$.

We have now described how linear functions, polynomials, as well as very non-linear and non-polynomial functions can be used to model data even while working with linear ansatze in w. This clearly is very promising; extremely complex data can be modeled using these ansatze provided one is artful enough to identify the appropriate ϕ_j. At the same time, this also presents an important dilemma which is most clearly visible in the polynomial ansatz (3.8). How does one choose the degree of M? For instance, a degree 1 polynomial is nothing but a degree 2 polynomial with $w_2 = 0$ or a degree 300 polynomial with $w_{300}, w_{299}, \ldots, w_2 = 0$. Would it be a good strategy to choose a polynomial of a high enough degree with the hope that if the data is truly linear, for example, then higher coefficients will automatically be set to zero? The next two sections are devoted to addressing precisely such questions [4].

3.2 Bias and Variance

Recall that the overall goal of machine learning is generalizability, i.e. to make good predictions for data which has not been seen yet. At the same time, it must be able to do so by learning from a finite amount of possibly, almost certainly, noisy data. We now try to fulfil these requirements while working in the toy setting of fitting data $\mathcal{D}_{train} = \{(x_\alpha, t_\alpha)\}$ to a univariate polynomial under the restrictions of Section 2, i.e. the performance on a putative test set $\mathcal{D}_{test}$ must be similar.

For concreteness, let us imagine that we have data (x, t) where t is the sum of an underlying regularity in x and a random noise. We will assume that the underlying regularity arises from a degree 3 polynomial and the noise is sampled from a normal distribution of mean 0 and standard deviation 1, i.e.

$$t = 0.5\,x^3 + 0.3\,x^2 - 0.25\,x + 1 + w \, , \quad w \sim \mathcal{N}(0, 1) \; . \tag{3.12}$$

This sample data is plotted in Figure 2 as a scatter plot along with a line plot of the underlying

[4] There is also an interesting parallel to neural networks which we would like to point out. Like polynomials, neural networks also come with a *universal approximation theorem*. This guarantees that a neural network with an arbitrarily large number of parameters may be brought arbitrarily close to any function of our choosing, provided the function obeys certain defined criteria. The corresponding statement for polynomials is the *Stone-Weierstrass Theorem* which guarantees that any function over a compact interval can be approximated by a polynomial. We will shortly see that the polynomial of very high degree are indeed capable of modeling very rich data. However, in order to prevent them from learning from noise in the training data, they have to be *regularized* as described in Section 3.3. A very close analogue of this phenomenon is found in neural networks where a very successful strategy for identifying neural network architectures to solve a dataset is to identify architectures which overfit by design on the training data, and then regulate them to ameliorate the overfitting.

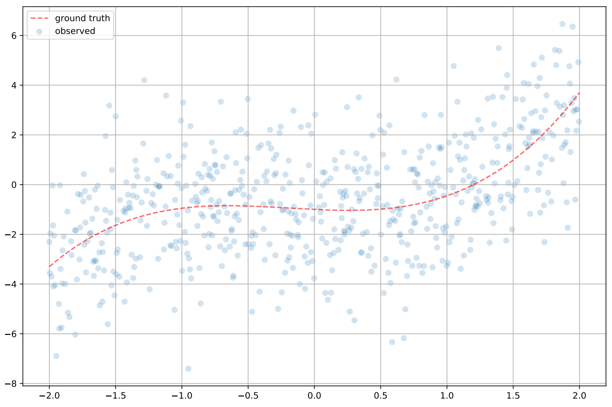


Figure 2: Noisy Data (scatter) with underlying regularity (red dashed line).

Degree	Train Loss	Test Loss
1	4.11	4.88
3	3.63	4.15
5	3.61	4.12
300	3.24	5.12×10^3

Table 1: Training and Test Losses for different degree polynomials used to model data of Figure 3.

regularity. We see that the data is quite noisy. We will now attempt to model the underlying regularity using ordinary least squares regression by choosing polynomial ansatze of degree 1, 3, 5 and 300. As mentioned at the outset, the goal of the program is to make good predictions on unknown data by learning from a finite amount of available data, possibly corrupted by noise. To estimate the performance of the the machine learning algorithms we will randomly set aside a *test set* and treat the remaining part of data as the *training data*. That is, the training data will be used to determine the optimal values of w for a given ansatz. This is called training the model. Once the model is trained, we will evaluate the model on the test data.

The results of this experiment are shown in Table 1, where we have recorded the training and test losses for the ansatze of different degrees at the end of training. On comparing the training losses we see that the degree 1 polynomial performs the worst, while the degree 300 polynomial performs the best. However, recall that the goal is to generalize to new data, for which our estimator is the test loss. Here we see that the degree 300 polynomial performs terribly. The test loss is three orders of magnitude larger than both the corresponding training loss and in fact any other loss on the table. The degree 3 and degree 5 polynomials perform comparably well, if anything, the degree 5 polynomial seems to do slightly better. At this point, one might be tempted to declare victory. We have compared across a class of models and found the best one. In principle, one could compare across a wider class of models, even going beyond polynomial regression, and find the best one and use that for future predictions. However, understanding *why* the linear and the degree 300 polynomials fail in different ways actually leads to interesting insights which are quite generally applicable to machine learning.

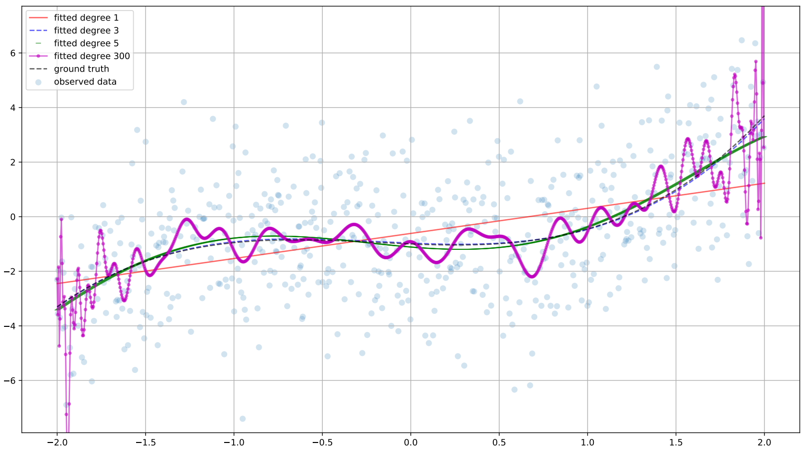


Figure 3: Ordinary Least Squares fitting against polynomials of different degree

Firstly, it is intuitively quite clear why the degree 1 polynomial performs poorly even for the training data iself. The underlying data which the polynomial is expected to model is nonlinear and further, corrupted by noise. It is almost expected then that a degree 1 polynomial will be unable to model the data with any great felicity. Such models are said to have *high bias*, i.e. the underlying assumptions in the model do not match well with the data itself. In contrast, the degree 300 polynomial with its 301 undetermined parameters has much more 'wiggle room'. Each of these parameters may be tuned independently to match with the training data as well as possible. This explains the fact that this model has the lowest training loss. However, by the same token, the model will also have a tendency to match as well as possible with the available training data even at the expense of learning irrelevant features which are purely artefacts of noise. Such features would expectedly be absent in the test data and the model would perform worse. Indeed from inspecting the learnt degree 300 polynomial plotted in Figure 3 one suspects that this is precisely what is happening; the polynomial appears to quite clearly chase after random fluctuations in data. Further, if we train such a model on a different training set, it is quite likely that it would again seek to mimic random fluctuations in *that* training set. The predictions drawn from the model when trained on different training data would also be likely to be significantly different to each other. Such models are said to have *high variance*, i.e. the model predictions vary significantly as the training data is changed. This is likely on account of the model having learnt spurious correlations in the training data. The poor performancy on the test set is then expectedly because these correlations are obtained from noise, and are absent – or different – in the test data.

To sum up, there are at least three sources of error when trying to apply a model to make predictions. Firstly, there is *intrinsic noise* in the data itself. These are random fluctuations in the data itself which by their nature are unpredictable or unlearnable [5]. Secondly, there is the model bias and thirdly the variance, both of which we have discussed. These are model dependent and therefore more directly controllable.

[5]However, we will see that in the Bayesian treatment of regression recounted in Appendix B, it is possible to estimate the parameters of a putative statistical distribution for the noise.

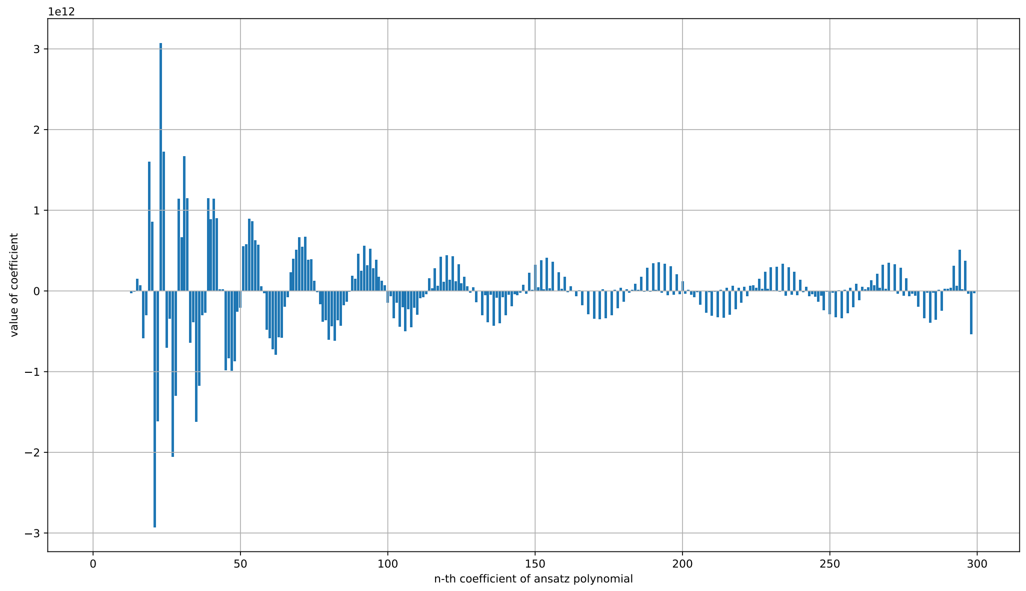


Figure 4: Coefficients of the degree 300 polynomial fixed by ordinary least squares fitting. Note the 10^{12} scale on the y axis.

It is intuitive that to have a good model with minimal *generalization error*, it is necessary to decrease both the bias and the variance. However, it is also intuitive that these two criteria are somewhat in tension with each other. A model with low bias, say deriving from an ansatz class which is rich enough to express complex data, is also likely to suffer from high variance. That is, its expressivity can also make it sensitive to random fluctuations in training data. This tension between bias and variance is called the *bias-variance tradeoff*. The phenomenon of the model performing significantly worse on the test data as compared to the training data is called *overfitting*. Conversely, it may also happen that the model actually performs worse on the training data itself. This is called *underfitting*, and typically indicates that we could choose a more complex model instead.

3.3 Regularization

The bias-variance tradeoff provides an important insight into model building, but also seems to lead us to an impasse. Driving down the bias invariably seems to increase the variance. How then do we build expressive models that express data in its full richness but prevent it from learning spurious correlations that arise purely from noise? Let us open up the trained degree 300 polynomial a bit more to get some more insight into this question. The coefficients of the trained polynomial are plotted out in Figure 4. We see that the coefficients are extremely large, some of the order of 10^{12} which likely are tuned to cancel off delicately on the training data.

One way to mitigate this fine-tuning is to discourage the coefficients from growing very large. This is achieved by *regularization*, where we add to the mean square cost function (3.5) additional

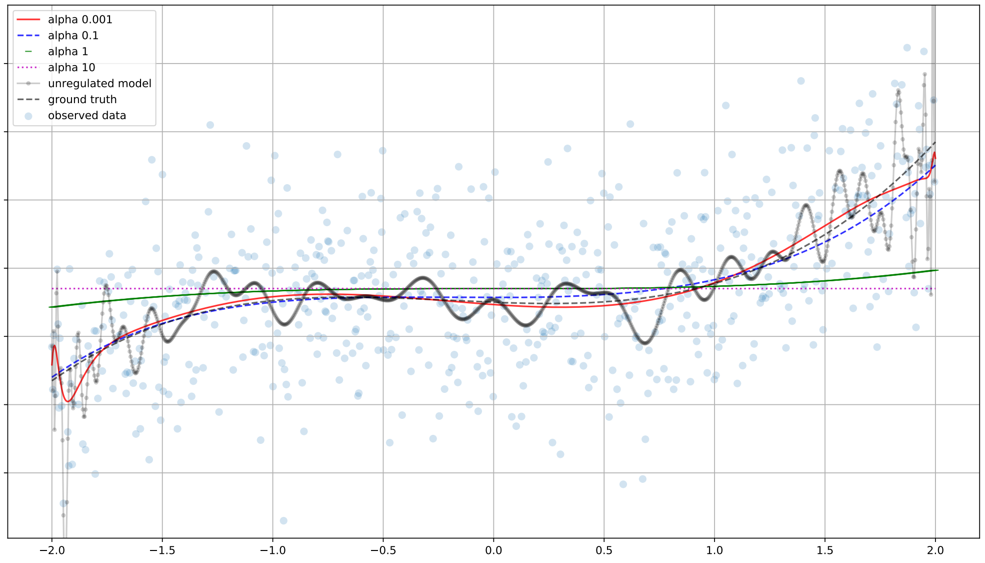


Figure 5: Coefficients of the degree 300 polynomial fixed by ordinary least squares fitting and L1 regularization. Small values of λ_1 provide solutions that mimic the underlying regularity.

terms which discourage the coefficients w from growing too large. The typical choice is

$$E(w) = \frac{1}{2}\frac{1}{N}\sum_{\alpha=1}^{N}\left(y(x_\alpha; w) - t_\alpha\right)^2 + \lambda_1\sum_{j=1}^{M}|w_j| + \lambda_2\sum_{j=1}^{M}w_j^2\,. \tag{3.13}$$

The terms multiplying λ_1 and λ_2 are the L1 and L2 norms of the vector w. The case where only λ_1 is non-zero is called L1 or Lasso regularization and where only λ_2 is non-zero is called L2 or Ridge regularization. Note that w_0 is typically excluded from the regularization term. The case where both λ_1 and λ_2 are non-zero is called elastic net.

Adding such terms forces the model to train while keeping the weights w as small as possible. In particular, though λs may be tiny, say $\mathcal{O}\left(10^{-4}\right)$ it would still discourage the model from learning coefficients that are of the order of 10^{12}. A standard practice is to deform the cost function away from (3.5) to (3.13) with small values of λs which are chosen experimentally. Figure 5 shows the effect of regularizing the degree 300 polynomial with different values of λ_1. Figure 6 shows coefficients of the degree 300 polynomial at a particular value of λ_1. Though we have motivated regularization as a convenient deformation of an intuitively obvious cost function (3.5) to achieve a desired end, it also seems a very *ad hoc* procedure. In particular, it was intuitively obvious what the loss function (3.5) does. It forces the ansatz polynomial to reproduce known data as well as possible. While we have seen that this is not the sole criterion for a good machine learning solution to a problem, and may in fact even be badly misleading, it still remains that any candidate machine learning solution must do well on known data. How do we interpret the L1/L2 regulators in (3.13)? To do so, it is instructive to adopt a somewhat different perspective than the one taken thus far. Our starting point is a hypothesis regarding the model which should describe observed data, e.g.

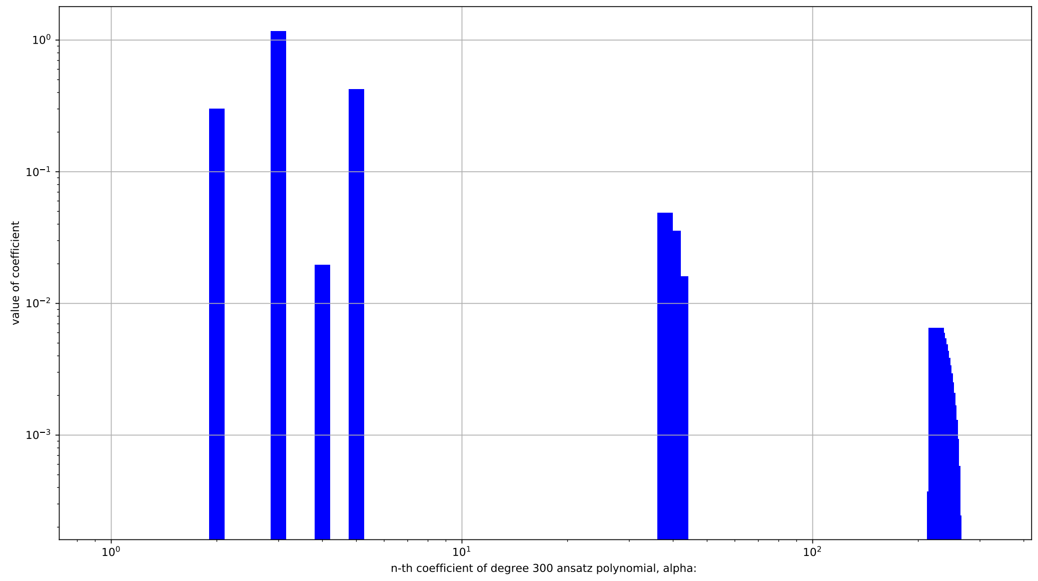


Figure 6: Coefficients of the degree 300 polynomial fixed by ordinary least squares fitting and L1 regularization.

we may postulate that the function that best captures the underlying trend in the data lies in the set of degree 300 polynomials. Given such a hypothesis, the regularization terms essentially enforce our *prior beliefs* about our model; that the coefficients are not, or should not be, large numbers. The data are on the other hand *evidence* for the parameters assuming some particular values. Optimizing the cost function (3.13) is the process of determining the parameters w guided by both prior expectations and the observed evidence. This line of thought feeds directly into the Bayesian approach to the same problem and has a strong parallel with the *maximum a posteriori* (MAP) principle, detailed in Appendix B.2.

The discussion above indicates that one way to develop expressive models that are resistant to overfitting is to start with models that are rich enough to capture the underlying complexities of data. Conversely, these models would also likely be prone to overfitting to the training data. We may then ameliorate this overfitting by regulating the model.

4 Optimization

The cost function (3.5) or its regularized counterpart (3.13) are minimized at some value w_*. We will now compute this value explicitly. It turns out that even though a formal solution for w_* is possible to construct, its numerical computation can be prohibitively difficult. For more complex cost functions, even a formal solution is impossible to write. Instead we turn to gradient-based methods for optimizing the cost function, which are the workhorses of deep learning. We will also describe a continuum approximation of these methods which is useful in some analytical studies.

To minimize the cost function (3.5), we note that it is quadratic in w for the GLMs, and computing partial derivatives with respect to ws gives us M equations, written below.

$$\partial_{w_i} E(w) = \sum_{\alpha=1}^{N} \left(t_\alpha - w \cdot \phi(x_\alpha)\right) \phi_i(x_\alpha) \; . \tag{4.1}$$

To proceed further, we define the *design matrix* Φ as

$$\Phi_{\alpha j} = \phi_j(x_\alpha) \; . \tag{4.2}$$

Then, defining t via $t^T = (t_1, t_2, \dots, t_N)$,

$$\sum_{n=1}^{N} t_\alpha \phi_i(x_n) = \sum_{\alpha} \Phi_{\alpha i} t_\alpha = \Phi^T \cdot t \tag{4.3}$$

and

$$\sum_{j,\alpha} w_j \phi_j(x_\alpha) \phi_i(x_\alpha) = \sum_{j,\alpha} w_j \Phi_{\alpha j} \Phi_{\alpha i} = \sum_{j,\alpha} \Phi^T_{i\alpha} \Phi_{\alpha j} w_j = \left(\Phi^T \Phi\right) \cdot w \, . \tag{4.4}$$

Hence the extremum of $E(w)$ is located at

$$w_* = \left(\Phi^T \Phi\right)^{-1} \Phi^T \cdot t \, . \tag{4.5}$$

Here $\left(\Phi^T \Phi\right)^{-1} \Phi^T$ is known as the *Moore-Penrose pseudo-inverse* of Φ. Note that even though the optimization can be carried out analytically, the computational complexity of the procedure scales as $\mathcal{O}\left(M^2\right)$ for with the number of features M, which is quite expensive. The complexity in the number of data N is better, scaling as $\mathcal{O}(N)$ but for large amounts of training data, the procedure may not fit in memory.

Thus we see that even relatively simple models which can be formally solved with pen and paper may be difficult to optimize in practice, simply because of the numerical cost of implementing the solution for data with many features or instances. For more complex cases like neural networks, the dependence of the cost function on the model parameters w is highly nonlinear and even a formal optimization as above is prohibitive. For both these reasons, we typically opt for methods that can be systematically implemented on the computer.

4.1 Gradient Descent

This is a generic optimization algorithm capable of finding local minima of given differentiable functions, e.g. cost functions of an ML problem. The idea for this algorithm dates back to Cauchy. The main steps involved in this algorithm are:

1. Start with an initial point w_i and compute $\nabla_w E(w)\,|_{w_i}$
2. Step in the direction of steepest descent (i.e. the negative gradient) by an amount η, called the *learning rate*.
$$w_{i+1} = w_i - \eta \, \nabla_w E(w_i) \; . \tag{4.6}$$
3. Replace w_i by w_{i+1} until the minimum is reached.

The learning rate is a *hyperparameter* of the problem. Choosing an optimal value is key to the success of the model. A very small value of η means the model can take too long to converge, and a very large value means the training may not converge, and indeed may even blow up.

4.1.1 Stochastic Gradient Descent

Note that the expression for $E(w)$ is a point-wise sum taken over each data appearing in the training set.

$$E(w) = \frac{1}{N}\sum_{\alpha=1}^{N} E(x_\alpha; w) \ . \tag{4.7}$$

This is often the case in machine learning. Computing $\nabla_w E(w)$ would then require us to sum over each point in the training set and can be expensive, and for a large enough N the operation may not even fit in memory.

Instead, at each step of gradient descent we approximate the gradient as

$$\nabla_w E(w) \simeq \frac{1}{m}\sum_{\alpha=1}^{m} \nabla_w E(x_\alpha; w) \ , \tag{4.8}$$

by summing only over m randomly chosen elements from N. In this manner we cycle through the training data, drawing m elements at a time until the entire training data has been exhausted. This is one *epoch* of training. We then repeat for as many epochs as needed to approach the minimum. When $m = 1$ the algorithm is called Stochastic Gradient Descent, and when $1 < m < N$ it is called minibatch Gradient Descent. m is then called the minibatch size.

Choice of m is a new hyperparameter of the problem and can also affect learning performance. This is especially true in the case of neural networks, where non-convexity of the cost function in w means that gradient descent can, and does, get trapped in local minima because the gradient vanishes. However, the gradient over a minibatch need not vanish and therefore there is a chance that the optimization algorithm can escape this false vacuum. Conversely though, for the same reason, SGD will not converge to the global minimum even for convex functions, and instead will fluctuate around it even at the end of training.

4.2 Gradient Flows

Gradient Descent admits a continuum limit which is useful for analytical study of neural networks. We have the update rule

$$w_{j+1} = w_j - \eta\, \nabla_w E(w_j) \ , \tag{4.9}$$

Define an new function $W(t) : \mathbb{R}_+ \to \mathbb{R}^{M+1}$ such that the w_j are samples from it at j timesteps seperated by η, i.e.

$$W(j\eta) = w_j \, , \qquad \forall \qquad t = j\eta \, . \tag{4.10}$$

Then we can write the above update rule as

$$W(t+\eta) = W(t) - \eta\, \nabla_W E(W(t)) \ , \tag{4.11}$$

i.e.

$$\frac{W(t+\eta) - W(t)}{\eta} = -\nabla_W E(W(t)) \ . \tag{4.12}$$

Taking the limit $\eta \to 0$,

$$\dot{W}(t) = -\nabla_W E(W(t)) \ , \tag{4.13}$$

where the overdot denotes differentiation with respect to t. This equation is the continuum limit of gradient descent and is therefore called a gradient flow.

5 Classification *via* Logistic Regression

We now turn to *classification*, i.e. learning to assign one of K class labels C_k to a given datum x on the basis of N previously supplied examples $\mathcal{D} = \{(x_n, C_n)\}$. A prototypical example is assigning a given $m \times n \times 3$ RGB image x containing a picture of a human to one of K people.

5.1 Binary Classification

We start with the case of $K = 2$, i.e. organizing data into classes $\mathcal{C}_1$ and $\mathcal{C}_2$. To set up this problem we first need to provide a numerical encoding for class labels. For binary classification we may define a target variable $t \in \{0, 1\}$ such that

$$t = \begin{cases} 1 & \Rightarrow \quad x \in C_1, \\ 0 & \Rightarrow \quad x \in C_2 \end{cases} . \tag{5.1}$$

In principle, one could use these target variables along with the 0-1 loss introduced in (2.4) variables to train a machine learning algorithm, much as the mean squared error (2.5) was used for the regression problem. However, this loss function is not very convenient to use along with gradient descent as it changes discontinuously [6].

In practice, therefore, another solution is more convenient. We will instead train a model that outputs $t \in (0, 1)$ which we interpret as $p\left(C_1|x\right)$. Next, given the output $p\left(C_1|x\right)$, we next need to identify a threshold value p_0 on the basis of which we declare that

$$x \in C_1 \quad \text{if} \quad t\left(x\right) > p_0\,, \quad \text{else} \quad x \in C_2\,. \tag{5.2}$$

A default choice for p_0 might be 0.5, though the actual value might depend on the application. As an example, 0.5 might be an unacceptably low threshold for declaring someone cancer free.

We now turn to modeling t for the binary classification problem. At least formally we may start with the *prior probabilities* $p\left(C_1\right)$ and $p\left(C_2\right)$ which represent our initial beliefs that a datum x would belong to class C_1 or C_2. Next, we have the *evidence* that the datum is takes value x. Hence, using Bayes theorem, the *posterior probability* $p\left(C_1|x\right)$ is

$$\begin{aligned} p\left(C_1|x\right) &= \frac{p\left(x|C_1\right)p\left(C_1\right)}{p\left(x\right)} = \frac{p\left(x|C_1\right)p\left(C_1\right)}{p\left(x|C_1\right)p\left(C_1\right) + p\left(x|C_2\right)p\left(C_2\right)} \\ &= \frac{1}{1 + \frac{p(x|C_2)p(C_2)}{p(x|C_1)p(C_1)}}\,. \end{aligned} \tag{5.3}$$

Then if we define

$$a = \frac{p\left(x|C_1\right)p\left(C_1\right)}{p\left(x|C_2\right)p\left(C_2\right)}\,, \tag{5.4}$$

[6] We will shortly define a loss function called the cross-entropy loss which may be trained *in lieu* of the 0-1 loss. However, it is useful to also keep an account of the expected value of the 0-1 loss as per (2.7) for the train and test data. It should be clear that (2.7) evaluated on the 0-1 loss simply measures one minus the *accuracy* of the classification, which is a very useful metric that is often tracked along with the value of the cross-entropy loss as the classifier trains or is evaluated on known test data.

and the *logistic sigmoid* function [7]

$$\sigma(a) = \frac{1}{1+e^{-a}}\,, \tag{5.6}$$

the posterior probability may be expressed as

$$p(C_1|x) = \sigma(a(x))\ . \tag{5.7}$$

So far this is a formal rewriting. But now we are free to model t, or $p(C_1|x)$, by choosing an ansatz for a in x with tunable parameters w. We choose

$$a(x) = \sum_{j=0}^{M} w_j \phi_j(x)\ . \tag{5.8}$$

This is a linear ansatz, much as the GLMs for regression studied previously. All the remarks made above (e.g. feature vectors, bias/variance tradeoff etc) apply equally well here. The weights w are to be determined by optimizing a loss function, defined below. This framework is called *logistic regression.*

In order to draw inferences from this framework, we have to define the threshold probability p_0 and the *decision boundary.* The latter is the locus of points which correspond to a fixed value of t and hence $p(C_1|x)$. This is clearly given by the hyperplanes

$$w\cdot\phi = \text{const} \tag{5.9}$$

in ϕ space. Logistic regression is therefore a *linear classifier*, as it learns linear decision boundaries. However, it can learn non-linear decision boundaries in x, as ϕ is in general a non-linear function of x.

We next turn to defining the cost function to train the model using Gradient Descent or its variants. In principle we can train the model using the mean square loss function, and indeed this is sometimes done in practice as well, even for neural networks. However, for most applications this is not the optimal choice. To see why, define

$$E(w) = \frac{1}{2N}\sum_{\alpha=1}^{N}(t_\alpha - \sigma_\alpha)^2\ , \qquad \sigma_\alpha \equiv \sigma(w\cdot\phi(x_\alpha))\ . \tag{5.10}$$

Then

$$\nabla_w E(w) = \frac{1}{N}\sum_{\alpha=1}^{N}(t_\alpha - \sigma_\alpha)\,\sigma'_\alpha \phi(x_\alpha)\ . \tag{5.11}$$

Recall that

$$\sigma'(a) = \sigma(a)(1-\sigma(a)) \approx 0\,, \tag{5.12}$$

when $\sigma \approx 0$ or 1. Hence learning would be very slow when the sigmoid function *saturates*, i.e. is very confident, even if it is wrong. This is far from desirable.

[7]Note two important properties of this function.

$$\sigma(-a) = 1-\sigma(a)\ , \qquad \sigma'(a) = \sigma(a)(1-\sigma(a))\ . \tag{5.5}$$

Instead, we define the *cross-entropy loss function* [8]

$$E\left(w\right)=-\frac{1}{N}\sum_{\alpha=1}^{N}\left(t_\alpha\log\sigma_\alpha+\left(1-t_\alpha\right)\log\left(1-\sigma_\alpha\right)\right)\ . \tag{5.13}$$

Now

$$\nabla_w E\left(w\right)=\frac{1}{N}\sum_{\alpha=1}^{N}\left(t_\alpha\frac{\sigma'_\alpha}{\sigma_\alpha}+\left(1-t_\alpha\right)\frac{\sigma'_\alpha}{1-\sigma_\alpha}\right)\phi\left(x_\alpha\right)=\frac{1}{N}\sum_{\alpha=1}^{N}\left(t_\alpha-\sigma_\alpha\right)\phi\left(x_\alpha\right)\ . \tag{5.14}$$

The more incorrect the output of the neuron, the faster it will learn. This greatly speeds up learning in practice. The final input needed to train a logistic regression classifier is *regularization*. We have already seen that generalized linear models can overfit when the number of predictors is large. A similar problem manifests itself here as well. To see this, note that minimizing the cross-entropy function (5.13) as per (5.14) requires that not only are training data assigned their correct labels, the class probabilites encoded in t_α also saturate to 0 or 1. A logistic regression classifier can achieve this by letting the weights and biases go extremely large because

$$\lim_{\lambda\to 0}\sigma\left(\lambda\, z\right)=\Theta\left(z\right)\ . \tag{5.15}$$

i.e. as the weights and biases of the sigmoid become extremely large, it behaves more and more like a step function, and predicts *certainties*, i.e. outputs very close to 0 and 1 for all inputs. If we have a large number of features, i.e. M is large, then it could be possible that a hyperplane might linearly separate our data in the ϕ space. In such an event, the weights and biases could indeed scale to large numbers. This is a sign of overfitting. That is to say, the model is becoming unreasonably confident about its training data. As for the generalized linear regressors above, this overfitting can be mitigated by L1/L2 or Elastic-Net regularization. At this point we also make some observations that are useful in the context of neural networks.

1. We can think of logistic regression as being a neural network of one neuron. It processes a weighted sum of its inputs and 'fires' an output dependent on the weighted sum.

2. For every datum x_n, the contribution to the gradient $\nabla_w E\left(w\right)$ is proportional to the *error* $\delta\equiv t_n-\sigma_n$, i.e. the difference between the observed and the expected output. This is also true for regression with mean square error cost function, see (4.1), and will be the seed of *back-propagation* in neural networks.

3. That the derivative of the sigmoid tends to vanish as we move away from a narrow central band will also be important when designing neural networks. It will turn out to be closely connected to the *vanishing gradients* problem.

5.2 Multinary Classification

The above framework easily extends to multiple classes, i.e., the problem of classifying given data x into 1 of K possible classes. That is, we seek to model class probablities $p\left(C_k|x\right)$ for given data

[8]As emphasized in [5], there is no *a priori* reason to single out (5.13) or even its multinary generalization (5.22) as 'the' cross-entropy. Indeed, even the mean squared error (3.5) may be regarded as the cross-entropy between the empirically observed data distribution and a postulated Gaussian distribution. See Appendix C for more details. Nonetheless, it is conventional to refer to (5.13) as the cross-entropy loss and we will continue to do so in what follows.

x. The analysis is very similar to the previous binary classification but with some slight differences. In particular, previously we were attempting to predict a single quantity, which was interpreted as $p(C_1|x)$ with it being implicit that $p(C_2|x) = 1 - p(C_1|x)$. Here, even for the $K = 2$ case there are two target variables, namely $p(C_1|x)$ and $p(C_2|x)$ with the constraint that $p(C_1|x) + p(C_2|x) = 1$ being automatically built in. We can of course rewrite the expressions here for $K = 2$ to recover the previous analysis.

We begin by returning to Bayes' Theorem as before to write

$$\begin{aligned} p(C_k|x) &= \frac{p(x|C_k)\,p(C_k)}{p(x)} \\ &= \frac{p(x|C_k)\,p(C_k)}{\sum_j p(x|C_j)\,p(C_k)}\,. \end{aligned} \tag{5.16}$$

We now define

$$a_k = \ln p(x|C_k)\,p(C_k)\ , \tag{5.17}$$

in terms of which

$$p(C_k|x) = \frac{\exp a_k}{\sum_j \exp a_j}\,. \tag{5.18}$$

The above function is called the normalized exponential or the softmax function. Next, we take the ansatz

$$a_k = w_k \cdot \phi\,, \tag{5.19}$$

in terms of which

$$p(C_k|x) = y_k(\phi) = \frac{\exp w_k \cdot \phi}{\exp \sum_j w_k \cdot \phi}\,. \tag{5.20}$$

We specify the target variables t_n for given data x_n as K-vectors with binary entries. In particular if x_n belongs to class C_j then all entries in t_n are zero, except for the j^{th} which is 1. In components,

$$t_{nk} = \delta_{jk} \qquad \Longleftrightarrow \qquad x_n \in C_j\,. \tag{5.21}$$

Next, the loss function for this classification is

$$E(w_1, \ldots, w_k) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln y_{nk}\,, \tag{5.22}$$

where $y_{nk} = y_k(x_n)$. At this point, it is also worthwhile to note that the above loss function, also called the *cross-entropy loss*, can be derived from a more probabilistic analysis which also paves the way for an eventual Bayesian treatment. Suppose we have observed data $\mathcal{D} = (x_n, t_n)$ which we seek to model through functions $y_k(\phi_n; w_1 \ldots w_k)$. Then the *likelihood* of observing this data is

$$p(\mathbf{t}|\mathbf{x}, w_1, \ldots, w_k) = \prod_{n=1}^{N}\prod_{k=1}^{K} p(C_k|\phi(x_n))^{t_{nk}} = \prod_{n=1}^{N}\prod_{k=1}^{K} y_{nk}^{t_{nk}}\,. \tag{5.23}$$

Here $\mathbf{x}, \mathbf{t}$ collectively denote the data x_n and t_n respectively. Each t_n is itself a K vector with components t_{nk}. Then one may postulate that the parameters w are such that this likelihood is maximized. Maximimizing this likelihood is equivalent to minimizing the negative log likelihood, which is just the cross entropy function (5.22) above. This framework of *maximum likelihood* *estimation* can just as well be applied to binary classification [9] and least squares regression to recover our previous analyses. See Appendix B.1 for details.

6 Neural Networks

Neural networks are a machine learning framework which draw inspiration from models of the human brain and biological learning. While it is a matter of some debate whether and to what extent does the analogy between biological and machine learning neural networks hold, it is undeniable that aspects of biological neural networks provide important intuition for the design and analysis of artificial neural networks. For this reason, even though we are almost ready to design a neural network with the inputs of the previous section, we will approach the problem from a different angle at least in the beginning. In particular we will draw inspiration from simple models of the human brain to construct machine learning frameworks.

6.1 The Perceptron

The fundamental structure in the human brain is a *neuron*. It receives stimuli and outputs a response based on the stimulus. The responses of some neurons may then be fed as stimuli to other neurons on the basis of which they produce responses. The simplest mathematical analogue of a single neuron is a *threshold logical unit* (TLU) which produces a binary output depending on whether or not the weighted sum of its inputs crosses a certain threshold.

$$f(x) = \Theta(w \cdot x + w_0) = \begin{cases} 0 : & w \cdot x + b < 0 \\ 1 : & w \cdot x + b > 0 \end{cases} . \tag{6.1}$$

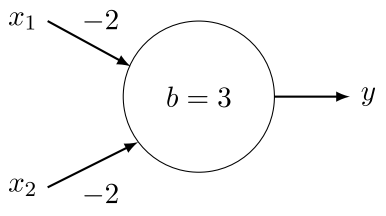
We can combine TLUs into a *fully connected layer* also known as a *dense layer*, i.e. take N TLUs and connect them to all the inputs. The resulting architecture is known as a perceptron [15]. Comparing (6.1) with (5.9) we see that the TLU is also a linear classifier, and its decision boundary is the locus

$$w \cdot x + b = 0 . \tag{6.2}$$

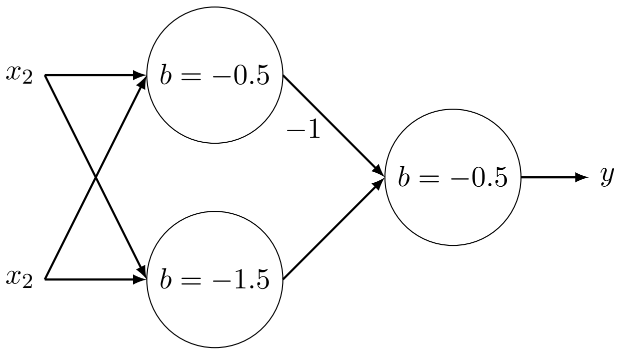
An important property of the TLU is that it can be used to build a network which functions as a *universal approximator*. This means that given any function $f(x)$ defined over some closed interval of the input data x, we can construct a network of TLUs whose output can be brought arbitrarily close to $f(x)$ for all values of x. By way of motivation, consider TLU shown in Figure 7a, which accepts binary inputs valued in $\{0, 1\}$. We can readily evaluate the input-output table for this TLU and verify that it is the truth table for the NAND gate, as shown in Table 2. Since the NAND gate is a universal logic gate, any logical circuit can be built purely from this TLU. Hence, if the function $f(x)$ is some logical function of the inputs x, we can realize it using a hypothetical TLU network. While this is true in principle, a key ingredient to actually harnessing universality properties is *depth*. To see this, consider the famous *XOR problem*, namely, how to design a perceptron which gives rise to the XOR truth table, shown in Table 3 [16]. It should be apparent by plotting these

[9] The likelihood function in that case would be

$$p(\mathbf{t}|\mathbf{x}, w_1, \ldots, w_k) = \prod_{n=1}^{N} p(C_1|\phi(x_n))^{t_n} (1 - p(C_1|\phi(x_n)))^{1-t_n} = \prod_{n=1}^{N} \sigma_n^{t_n} (1 - \sigma_n)^{1-t_n} . \tag{5.24}$$



(a) NAND Gate TLU, see Table 2



(b) XOR MLP. see Table 3. ws equal 1 unless otherwise stated.

x_1	x_2	Output
0	0	1
0	1	1
1	0	1
1	1	0

Table 2: Input/Output Table for the NAND TLU shown in Figure 7a.

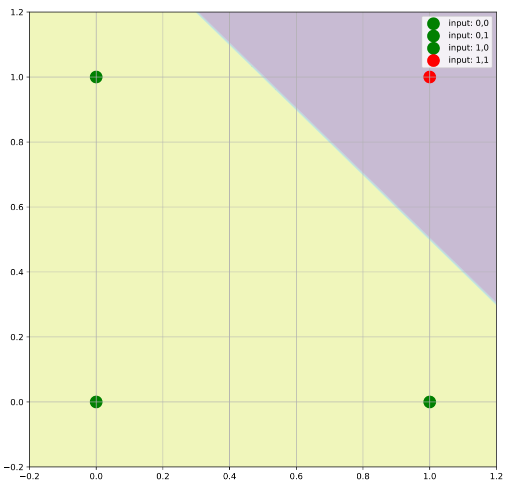
x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 3: Input/Output Table for the XOR TLU shown in Figure 7a.

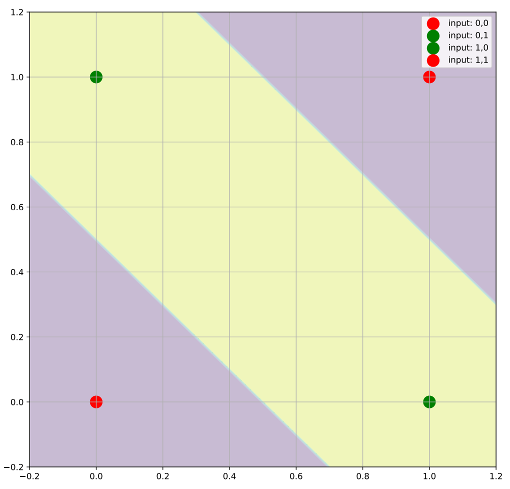
four points on the x_1, x_2 plane that points corresponding to this data is not linearly separable, i.e. there does not exist any single straight line such that $(0, 0)$ and $(1, 1)$ lie on one side of the line while $(0, 1)$ and $(1, 0)$ lie on the other side. See Figure 8b for a plot of these four points along with the decision boundary that *does* separate them. In contrast, the NAND input-outputs of Table 2 are indeed linearly separable, as shown in Figure 8a, and a single TLU can produce this data. It turns out that the key to solving the XOR problem is to incorporate a *hidden layer*, i.e. a set of TLUs which neither directly process an input, nor produce an output. Instead, they act as intermediaries between the input and output layers. For the XOR problem, it turns out that a single hidden TLU is enough. An architecture that realizes the XOR classification is shown in Figure 7b.

Universal Approximator for Functions We have seen that including a hidden layer allows us to learn the XOR classifier, which has a non-linear decision boundary. Since data in general has non-linearities, clearly incorporation of hidden layers is going to be crucial. To illustrate this point we now sketch an argument that any function over a compact support can be approximated arbitrarily well by a one hidden layer perceptron. This *universal approximation theorem* guarantees that neural networks can learn any function of our choosing [10]. We will do this for a univariate function $f(x)$ having support over the closed interval $I = [x_i, x_f]$, following [17]. In general the proof holds for multivariate, multiple output functions. We refer the reader to this link for an interactive visualization of this proof. To make contact with the classification and regression problems studied above, we assume that $f(x)$ could be a regression function that we are trying to predict, or that it

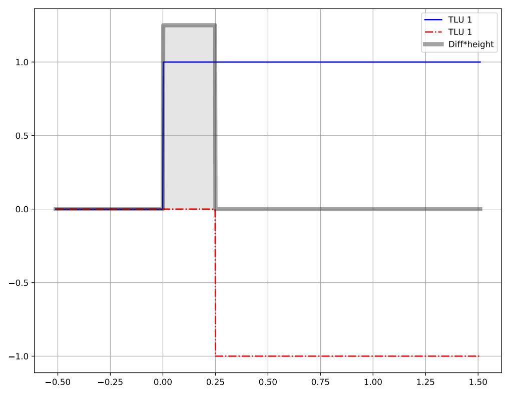
[10]This is a geometric proof which relies quite heavily on the fact that the activation function is a step function. The proof for the sigmoid activation is more formal and tend to involve proving some version of the statement that the set of functions that can be learnt by sigmoid activated neurons in a neural network having a single hidden layer of infinite width is dense in the set of functions, i.e. given any function, a neural network can be designed which is arbitrarily close to that function. Proofs for other activation functions have also been done.



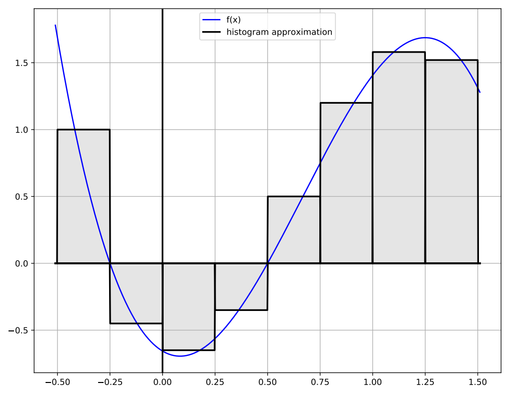
(a) Decision boundary for NAND classifier



(b) Decision boundaries for XOR classifier



(a) A rectangular function from TLUs



(b) Piecewise approximation of a function could be the decision boundary of a classifier, so that $p\left(C_1|x\right) = \sigma\left(f\left(x\right)\right)$. These assumptions are not relevant to the actual argument below.

First, divide the interval I into N equally spaced bins of width

$$w = \frac{x_f - x_i}{N}\,. \tag{6.3}$$

The starting and ending points of the j^{th} bin are x_{j-1} and x_j respectively, where

$$x_j = x_i + j \times w\,. \tag{6.4}$$

Consider the following approximation of $f\left(x\right)$

$$f_0\left(x\right) = f\left(\bar{x}_j\right)\,, \quad x \in \left[x_{j-1}, x_j\right]\,, \quad j = 1, 2, \ldots, N\,, \tag{6.5}$$

where

$$f\left(\bar{x}_j\right) = f\left(\frac{x_{j-1} + x_j}{2}\right)\,. \tag{6.6}$$

In general any other approximate value of $f\left(x\right)$ which is good over this $\left[x_{j-1}, x_j\right]$ interval will also do equally well.

By increasing N, f_0 can be brought arbitrarily close to f. The proof of the theorem consists of showing how (6.5) is realized by an MLP of one hidden layer. To do so, consider the TLU given by

$$f_s\left(x\right) = \Theta\left(wx + b\right)\,, \qquad s = -\frac{b}{w}\,. \tag{6.7}$$

We have indexed the functional form of the TLU by s, the threshold value at which it fires. Then, consider the function

$$f_{s_1,s_2,h}\left(x\right) = h\left(f_{s_1}\left(x\right) - f_{s_2}\left(x\right)\right)\,. \tag{6.8}$$

This has the values

$$f_{s_1,s_2,h}\left(x\right) = \begin{cases} h & : \quad x \in \left(s_1, s_2\right) \\ 0 & : \quad \text{otherwise} \end{cases}\,. \tag{6.9}$$

Then we can define

$$f_j\left(x\right) \equiv f_{x_{j-1},x_j,f\left(\bar{x}_j\right)}\left(x\right)\,. \tag{6.10}$$

Finally

$$f_0\left(x\right) = \sum_{j=1}^{N} f_j\left(x\right)\,. \tag{6.11}$$

Thus, we have realized the approximation (6.5) purely in terms of TLUs lying in one hidden layer. The outputs of the TLUs are all added to each other in the output layer.

From these discussions it appears that Multi-Layer Perceptrons (MLPs) are powerful frameworks, capable of expressing rich functions. However, they are quite difficult to train in practice. Since they are constructed out of step functions, gradient descent-like algorithms are difficult to deploy on them as the derivative of the step function is a Dirac delta, strictly zero in w and b except for a single point. Further, even tuning the weights can be challenging in MLPs since the output of the TLU doesn't change with small tunings, until it suddenly jumps. In a multi-layer architecture, this jump now causes the input to another TLU to jump. This makes the whole framework very difficult to control during training.

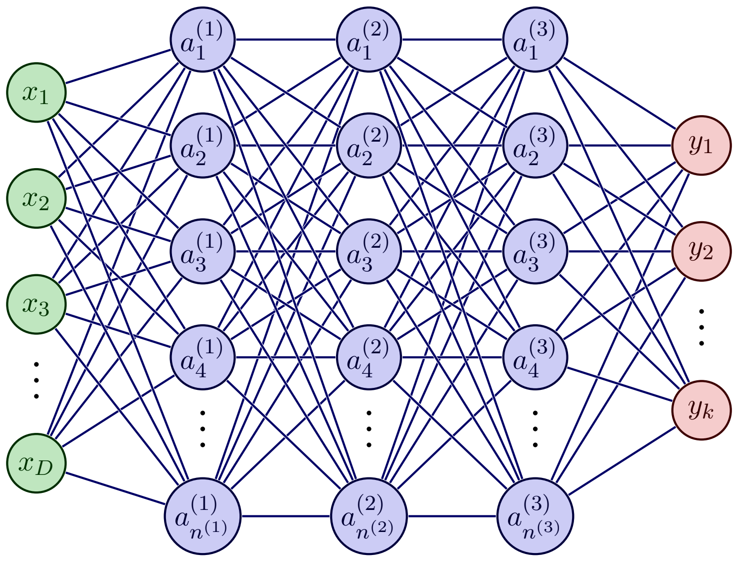


Figure 10: An artificial neural network of three hidden layers with k outputs.

It was subsequently realized that the MLP architecture could be deformed slightly to replace the TLUs with sigmoid functions. We recall that sigmoid functions are, loosely speaking, deformations of the TLU to continuous functions as seen in (5.15). This deformed network can be trained using gradient descent. The resulting architecture, first developed in [18], is the first modern neural network and we now turn to its study.

6.2 Artificial Neural Networks

We now turn to Artificial Neural Networks, also known as dense neural networks or deep neural networks. In some places, these are also called multi-layer perceptrons. Figure 10 shows an MLP with three hidden layers which processes D inputs x into k outputs y. The most straightforward way to describe this ANN is as a series of functional transformations.

1. Take the following $n^{(1)}$ linear combinations of the inputs

$$a_j^{(1)} = w_{ji}^{(1)} x_i + w_{j0}^{(1)} \,, \qquad j = 1, \ldots n^{(1)},\, i = 1, \ldots, D\,. \tag{6.12}$$

   These are called the *preactivations* of layer 1.

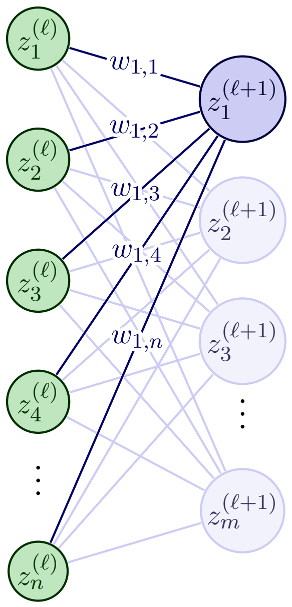
2. Transform each $a_j^{(1)}$ via a nonlinear activation function $h^{(1)}$ as

$$z_j^{(1)} = h^{(1)}\left(a_j^{(1)}\right)\,. \tag{6.13}$$

   These are the corresponding *activations* and are the outputs of layer 1.

3. The preactivations and activations of layer 2 are determined from the activations of layer 1 as above, replacing the data x with $z^{(1)}$ i.e.

$$a_k^{(2)} = w_{kj}^{(2)} z_j^{(1)} + w_{j0}^{(2)} \,, \quad z_k^{(2)} = h^{(2)}\left(a_k^{(2)}\right)\,, \quad k = 1, \ldots n^{(2)}\,. \tag{6.14}$$



$$\begin{pmatrix} z_1^{(\ell+1)} \\ z_2^{(\ell+1)} \\ \vdots \\ z_m^{(\ell+1)} \end{pmatrix} = h^{(\ell)} \left[\begin{pmatrix} w_{1,0} & w_{1,1} & \dots & w_{1,n} \\ w_{2,0} & w_{2,1} & \dots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,1} & \dots & w_{m,n} \end{pmatrix} \begin{pmatrix} z_1^{(\ell)} \\ z_2^{(\ell)} \\ \vdots \\ z_n^{(\ell)} \end{pmatrix} + \begin{pmatrix} b_1^{(\ell)} \\ b_2^{(\ell)} \\ \vdots \\ b_m^{(\ell)} \end{pmatrix} \right]$$

$$z^{(\ell+1)} = h^{(\ell)} \left(\mathbf{W}^{(\ell)} z^{(\ell)} + \mathbf{b}^{(\ell)} \right), \quad \mathbf{W}^{(\ell)} \in \mathbb{R}^{m \times n} .$$

Figure 11: The recursive determination of the outputs of layer $\ell + 1$ from those of layer ℓ in a Dense Neural Network, *aka* Fully-Connected Neural Network.

4. We recurse this structure, determining the preactivations and activations of layer ℓ from the activations of layer $\ell - 1$ as above.

5. Finally, given the outputs of the L^{th} hidden layer, which feeds into the output, we can compute the output preactivations and the outputs themselves.

$$a_k^{(L+1)} = w_{kj}^{(L+1)} z_j^{(L)} + w_{j0}^{(L)} , \quad y_k = h^{(\text{out})} \left(a_k^{(L+1)} \right) , \quad k = 1, \dots K . \tag{6.15}$$

   Here $h^{(\text{out})}$ could be the identity if we are doing regression or it could be the sigmoid or softmax for classification tasks.

The DNN would typically then be trained using the mean squared error or cross-entropy loss for regression or classification respectively.

Note also that if, for illustration's sake, we have one hidden layer and are doing classification so that $h^{(\text{out})} = \sigma$ then the outputs y can directly be written in terms of the inputs x as

$$y_k = \sigma \left(w_{kj}^{(2)} \phi_j(x) + w_{k0}^{(2)} \right) , \qquad \phi_j(x) = h^{(1)} \left(w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) . \tag{6.16}$$

Hence, one way of thinking about this neural network is that it does logistic regression while also automatically trying to construct optimal features to classify on. More generally, we see that the recursive structure of feeding the outputs of layer ℓ as inputs to layer $\ell + 1$ essentially means that the layer $\ell + 1$ learns on what the layer ℓ has processed. This leads to a heirarchical structure where deeper layers learn progressively more and more complicated structures from relatively simpler structures learnt by the previous layers. This is particularly apparent in the case of convolutional neural networks.

6.3 How Neural Networks Learn: Backpropagation

We have constructed a neural network with tunable parameters which we collectively denote by w. In order to train the network using gradient descent, we need to compute the gradients $\left\{\frac{\partial}{\partial w}E\right\}$ of the cost function E for all weights w, at every gradient descent step. Here E is the cost function. The computation of gradients is done by the backpropagation algorithm which we now describe. [11] Firstly note that typically error functions are a sum of terms, one for each element (x_α, t_α) in the training set $\mathcal{D}_{train}$.

$$E(w) = \sum_{\alpha=1}^{N} E_\alpha(w) \ . \tag{6.18}$$

We will concentrate on one term $E_n(w)$ and compute the gradients $\left\{\frac{\partial}{\partial w}E_n\right\}$. We can subsequently sum over the entire training batch to compute the full gradient or, more likely, sum over a minibatch to compute the minibatch gradient.

To fix our ideas it is useful to consider the L-layer MLP architecture described previously. This takes the form of the following equations

$$\begin{array}{ccccc} a^{(1)} = w^{(1)} \cdot z^{[0]} & a^{(2)} = w^{(2)} \cdot z^{[1]} & \ldots & a^{(L)} = w^{(L)} \cdot z^{[L-1]} & a^{(L+1)} = w^{(L+1)} \cdot z^{[L]} \\ z^{(1)} = h^{(1)}\left(a^{[1]}\right) & z^{(2)} = h^{(2)}\left(a^{[2]}\right) & \ldots & z^{(L)} = h^{(L)}\left(a^{[L]}\right) & y = h^{(L+1)}\left(a^{[L+1]}\right) \end{array} \ . \tag{6.19}$$

In components

$$y_k = \sum_{j=0}^{n^{(L)}} h^{(L+1)}\left(w_{kj}^{(L+1)} \cdot z_j^{[L]}\right) \ . \tag{6.20}$$

For concreteness we will consider a regression problem with given N data $\{(x_\alpha, t_\alpha)\}$ where $x \in \mathbb{R}^D$ and $t \in \mathbb{R}^K$. Then $h^{(L+1)} = id$, and we train the network using the mean square loss. The same backpropagation equations are obtained on studying classification problems with the cross-entropy loss function and $h^{(L+1)} = \sigma$. Still more general losses and activations also only require minimal modifications.

We have

$$E(w) = \frac{1}{N}\sum_{\alpha=1}^{N} E_\alpha(w) = \frac{1}{N}\sum_{\alpha=1}^{N}\sum_{k=1}^{K}\left(t_{k;\alpha} - y(x_\alpha; w)\right)^2 \ , \tag{6.21}$$

where

$$y(x_\alpha; w) = \sum_{j=0}^{n^{(L)}} w_{kj}^{(L+1)} \cdot z_j^{[L]}(x_\alpha) \ , \tag{6.22}$$

and is itself implicitly a function of the hidden layer parameters $w^L, w^{L-1} \ldots w^{(1)}$. We will compute the gradients of E_α with respect to the ws. These itself are stochastic approximations which can be used with a randomly chosen α for a gradient descent step. Minibatch approximations or the

[11] In principle it is possible to numerically approximate the gradient as

$$\frac{\partial}{\partial w}E(w) = \lim_{\epsilon\to 0}\frac{1}{\epsilon}\left[E(w+\epsilon) - E(w)\right] \qquad \forall \qquad w \in \{w_{ij}\} \ . \tag{6.17}$$

In practice however this is very slow. We would need to compute 2 forward passes W times, where W is the number of weights. In contrast, the backpropagation algorithm described here needs a single forward pass to compute all the derivatives needed for a gradient descent step. However, this naive procedure is useful for checking if our implementation of the backpropagation algorithm is correct.

full batch gradient can be computed by summing over α and normalizing by minibatch size or N as appropriate.

Firstly, note that computing gradients with respect to the weights $w^{(L+1)}$ is straightforward. We will write it in a way that makes the generalization to lower layers apparent.

$$\frac{\partial E_\alpha}{\partial w_{kj}^{(L+1)}} = \sum_{k=1}^{K} \left(t_{k;\alpha} - a_{k;\alpha}^{(L+1)}\right)^2 , \qquad a_{k;\alpha}^{(L+1)} = \sum_{j=0}^{n^{(L)}} w_{kj}^{(L+1)} \cdot z_j^{[L]}\left(x_\alpha\right) . \tag{6.23}$$

Then

$$\frac{\partial E_\alpha}{\partial w_{kj}^{(L+1)}} = \frac{\partial E_\alpha}{\partial a_{k;\alpha}^{(L+1)}} \frac{\partial a_{k;\alpha}^{(L+1)}}{\partial w_{kj}^{(L+1)}} = \left(a_{k;\alpha}^{(L+1)} - t_{k;\alpha}\right) z_j^{[L]}\left(x_\alpha\right) . \tag{6.24}$$

Next, since $y_{k;\alpha} = a_{k;\alpha}^{(L+1)}$,

$$\frac{\partial E_\alpha}{\partial w_{kj}^{(L+1)}} = \delta_{k;\alpha}^{(L+1)} z_{j;\alpha}^{(L)} , \qquad \delta_{k;\alpha}^{(L+1)} \equiv y_{k;\alpha}^{(L+1)} - t_{k;\alpha} , \tag{6.25}$$

where we have used $a_{k;\alpha}^{(L+1)} = y_{k;\alpha}$. The $\delta^{(L+1)}$ are known as the *errors* in the outer layer. In the above, j ranges from 0 to $n^{(L)}$. We can also write out the parameter $w_{k0}^{(L+1)}$ explicitly as the bias $b_k^{(L+1)}$. Hence, (6.25) may also be written as

$$\frac{\partial E_\alpha}{\partial w_{kj}^{(L+1)}} = \delta_{k;\alpha}^{(L+1)} z_{m;\alpha}^{(L)} , \qquad \frac{\partial E_\alpha}{\partial b_k^{(L+1)}} = \delta_{k;\alpha}^{(L+1)} , \qquad \delta_{k;\alpha}^{(L+1)} \equiv y_{k;\alpha}^{(L+1)} - t_{k;\alpha} . \tag{6.26}$$

We note that the identical expression is obtained on replacing the mean square loss with the cross entropy loss and the σ activation for $h^{(L+1)}$. Recall Equation (5.14) and the notes below it. We will still use the chain rule in the form $\frac{\partial E_\alpha}{\partial w_{kj}^{(L+1)}} = \frac{\partial E_\alpha}{\partial a_{k;\alpha}^{(L+1)}} \frac{\partial a_{k;\alpha}^{(L+1)}}{\partial w_{kj}^{(L+1)}}$. Still more generally, if we have a general loss function which is neither the mean square error nor the cross entropy, we can still compute

$$\frac{\partial E_\alpha}{\partial w_{kj}^{(L+1)}} = \delta_{k;\alpha}^{(L+1)} z_{j;\alpha}^{(L)} , \qquad \delta_{k;\alpha}^{(L+1)} \equiv \frac{\partial E_\alpha}{\partial a_{k;\alpha}^{(L+1)}} , \tag{6.27}$$

Computing the latter derivative just relies on the knowledge of the functional form of the cost function and can be done.

Claim: The gradients of the weights of the hidden layers have a very similar structure. In particular,

$$\frac{\partial E_\alpha}{\partial w_{kj}^{(\ell)}} = \delta_{k;\alpha}^{(\ell)} z_{j;\alpha}^{(\ell-1)} , \qquad \delta_{k;\alpha}^{(\ell)} \equiv \frac{\partial E_\alpha}{\partial a_{k;\alpha}^{(\ell)}} . \tag{6.28}$$

where $\delta_{k;\alpha}^{(\ell)}$ are yet to be computed explicitly. We will see that there is a very nice recursive structure in the δs which determines $\delta^{(\ell)}$ completely in terms of $\delta^{(\ell+1)}$. Let us turn to deriving this. We begin by noting that

$$E_\alpha\left(\ldots, w^{(\ell)}, \ldots\right) \equiv E_\alpha\left(\ldots, a^{(\ell)}\left(w^{(\ell)}\right), \ldots\right) , \tag{6.29}$$

i.e. the dependence of w^ℓ in $E_\alpha(w)$ comes from the preactivation

$$a^{(\ell)}_{k;\alpha} = \sum_{j=0}^{n^{(\ell-1)}} w^{(\ell)}_{kj} \cdot z^{[\ell]}_j (x_\alpha) \,. \tag{6.30}$$

A particular instance of this statement is our starting point for backpropagation, Equation (6.23), itself. There we explicitly, if somewhat trivially, see that $E_\alpha(w)$ depends on $w^{(L+1)}$ exclusively through $a^{(L+1)}$. As as result of (6.29), we have

$$\frac{\partial E_\alpha}{\partial w^{(\ell)}_{kj}} = \frac{\partial E_\alpha}{\partial a^{(\ell)}_{k;\alpha}} \frac{\partial a^{(\ell)}_{k;\alpha}}{\partial w^{(\ell)}_{kj}} = \frac{\partial E_\alpha}{\partial a^{(\ell)}_{k;\alpha}} z^{(\ell-1)}_{j;\alpha} \,. \tag{6.31}$$

This already verified the general structure of (6.23). Next, to determine $\delta^{(\ell)} \equiv \partial_{a^{(\ell)}} E$, we note that E_α depends on $a^{(\ell)}_{k;\alpha}$ only through the combinations $a^{(\ell+1)}_{k;\alpha}$, i.e.

$$\begin{aligned} a^{(\ell+1)}_{k;\alpha} &= \sum_{m=0}^{n^{(\ell)}} w^{(\ell+1)}_{km} z^{(\ell)}_{m;\alpha} \\ &= \sum_{m=0}^{n^{(\ell)}} w^{(\ell+1)}_{km} h^{(\ell)} \left[a^{(\ell)}_{m;\alpha} \right] \,. \end{aligned} \tag{6.32}$$

As a result

$$\frac{\partial E_\alpha}{\partial a^{(\ell)}_{j;\alpha}} = \sum_{k=0}^{n^{(\ell+1)}} \frac{\partial E_\alpha}{\partial a^{(\ell+1)}_{k;\alpha}} \frac{\partial a^{(\ell+1)}_{k;\alpha}}{\partial a^{(\ell)}_{j;\alpha}} \tag{6.33}$$

Using (6.32) and the definition of δ^ℓ we find

$$\delta^{(\ell)}_{j;\alpha} = h'_{(\ell)} \left[a^{(\ell)}_{j;\alpha} \right] \sum_{k=0}^{n^{(\ell+1)}} \delta^{(\ell+1)}_{k;\alpha} w^{(\ell+1)}_{kj} \,. \tag{6.34}$$

We therefore see that the errors $\delta^{(\ell)}$ for hidden neurons are completely determined by the errors $\delta^{(\ell+1)}$. Errors therefore propagate backward from the output layer into the network, giving rise to the name *backpropagation*. Hence we have implicitly determined all the derivatives required for a single gradient descent step in terms of the following equations.

$$\frac{\partial E_\alpha}{\partial w^{(\ell)}_{kj}} = \delta^{(\ell)}_{k;\alpha} z^{(\ell-1)}_{j;\alpha} \,, \qquad \delta^{(\ell)}_{j;\alpha} = \begin{cases} y_{k;\alpha} - t_{k;\alpha} \,, \quad \ell = L+1 \,, \\ h'_{(\ell)} \left[a^{(\ell)}_{j;\alpha} \right] \sum_{k=0}^{n^{(\ell+1)}} \delta^{(\ell+1)}_{k;\alpha} w^{(\ell+1)}_{kj} \end{cases} \,. \tag{6.35}$$

As before, this includes the $j = 0$, i.e. bias, term. Writing that out explicitly,

$$\frac{\partial E_\alpha}{\partial w^{(\ell)}_{kj}} = \delta^{(\ell)}_{k;\alpha} z^{(\ell-1)}_{j;\alpha} \,, \qquad \frac{\partial E_\alpha}{\partial b^{(\ell)}_{k}} = \delta^{(\ell)}_{k;\alpha} \,. \tag{6.36}$$

This expression can be straightforwardly modified to more general topologies. The sum over k now no longer runs over the level $\ell + 1$ but instead over all neurons k *to which* the neuron j sends connections. Similarly, the forward propagation equations (6.19) would also need to be modified.

6.4 Vectorized Backpropagation

The above expressions are implemented numerically in matrix form. This minimizes explicit function calls by handling the entire minibatch, indexed by α, in one go. In practice this is a much faster implementation than calling the above expressions for every datum $\{(x_\alpha, t_\alpha)\}$ individually. We define the matrices

$$\boldsymbol{W}^{(\ell)} \equiv \delta^{(\ell)}_{kj}\,, \quad \boldsymbol{\delta}^{(\ell)} \equiv \delta^{(\ell)}_{k;\alpha}\,, \quad \boldsymbol{z}^{(\ell)} \equiv z^{(\ell)}_{j;\alpha}\,, \quad \boldsymbol{a}^{(\ell)} \equiv a^{(\ell)}_{j;\alpha}\,. \tag{6.37}$$

Next, we compute the minibatch approximation to the gradients $\partial_w E$, given by

$$\frac{\partial E}{\partial w^{(\ell)}_{kj}} = \frac{1}{m_B}\sum_{\alpha=1}^{m_B} \delta^{(\ell)}_{k;\alpha} z^{(\ell-1)}_{j;\alpha}\,, \qquad \frac{\partial E}{\partial b^{(\ell)}_{k}} = \frac{1}{m_B}\sum_{\alpha=1}^{m_B} \delta^{(\ell)}_{k;\alpha}\,. \tag{6.38}$$

Writing this as a matrix equation,

$$\frac{\partial E}{\partial \boldsymbol{W}^{(\ell)}} = \frac{1}{m_B}\boldsymbol{\delta}^{(\ell)} \cdot \boldsymbol{z}^{(\ell)\,t}\,, \qquad \frac{\partial E}{\partial \boldsymbol{b}^{(\ell)}} = \frac{1}{m_B}\sum_{\alpha} \boldsymbol{\delta}^{(\ell)}\,. \tag{6.39}$$

Here

$$\boldsymbol{\delta}^{(L+1)} = \boldsymbol{y} - \boldsymbol{t}\,, \qquad \boldsymbol{\delta}^{(\ell)} = \left[\boldsymbol{W}^{(\ell+1)\,t} \cdot \boldsymbol{\delta}^{(L+1)}\right] * h'_{(\ell)}\left[\boldsymbol{a}^{(\ell)}\right]\,. \tag{6.40}$$

Here $*$ is the Hadamard product of matrices, defined by

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22}\end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22}\end{pmatrix} = \begin{pmatrix} a_{11}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{22}b_{22}\end{pmatrix}\,, \tag{6.41}$$

and h' is applied point-wise as usual

$$h'_{(\ell)}\left[\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22}\end{pmatrix}\right] = \begin{pmatrix} h'_{(\ell)}\left[a_{11}\right] & h'_{(\ell)}\left[a_{12}\right] \\ h'_{(\ell)}\left[a_{21}\right] & h'_{(\ell)}\left[a_{22}\right]\end{pmatrix}\,. \tag{6.42}$$

These expressions can be implemented numerically to train a neural network.

7 Training a Neural Network

We have defined a neural network architecture, and provided a prescription for learning *via* gradient descent. As things stand, this should be enough to start training neural networks effectively. However, a naive application of this procedure has long been known to be plagued with difficulties. In this section we will outline some difficulties in training multi-layer (i.e. deep) neural networks and how these may be mitigated. This will also give rise to some additional building blocks of deep neural networks, as well as some modern best practices for training neural networks.

7.1 The Vanishing Gradients Problem

For a dense neural network, the expression (6.34) can be used to write down the expression for errors in the p^{th} layer, $\delta^{(p)}_j$ as

$$\delta^{(p)}_j = h'\left(a^{(p)}_j\right)\sum_k \delta^{(p+1)}_k w^{p+1}_{kj}\,. \tag{7.1}$$

Then, using the backprop framework we can recursively write expressions for the errors in the $L, L-1, L-2, \ldots$ layers as [12]

$$\begin{aligned}
\delta_j^{(L)} &= h'\left(a_j^{(L)}\right) \sum_k \delta_k^{(\text{output})} w_{kj}^{(L+1)} \\
\delta_j^{(L-1)} &= h'\left(a_j^{(L)}\right) \sum_k \delta_k^{(L)} w_{kj}^{(L-1)} \sim \left(h'\right)^2 \\
\delta_j^{(L-2)} &= h'\left(a_j^{(L-1)}\right) \sum_k \delta_k^{(L-1)} w_{kj}^{(L-2)} \sim \left(h'\right)^3 \,,
\end{aligned} \tag{7.2}$$

and so on. Now the sigmoid activation function $\sigma(z)$ has mostly very small derivatives away from a small region about $z = 0$ as shown in Figure 13 below. From the above order of magnitude estimates, the errors (and hence the corresponding gradients) rapidly become tiny as we move back in layers. Hence, gradient descent is unable to change the weights of the inner layers by any appreciable amount. This is known as the *vanishing gradients problem.* More generally, training deep neural networks suffers from the *unstable gradients* or even the *exploding gradients* problem. We next examine how the choice of initialization prescription and activation functions can be used to ameliorate the vanishing gradients problem.

7.1.1 Initialization

Recall that Gradient Descent, or its variants and extensions, are all *local processes.* They start at an initial point in weight space, and iteratively update coordinates to try and minimize the cost function. Choosing the initial configuration is clearly very important in this whole process. A simple choice might be to initialize all the weights and biases to zero. However, this is not an optimal choice. This is because the neurons in a given layer ℓ of an MLP are all indistinguishible from each other, see the forward and backward propagation equations. The only way to break the $n^{(\ell)}$-fold degeneracy within a layer is to initialize weights randomly by sampling them from a statistical distribution. Two popular choices are the Normal distribution $\mathcal{N}(0, \sigma)$ and the Uniform distribution $U(-a, a)$ both centred about zero. For a long time, the extant practice was to initialize the weights in say $\mathcal{N}(0,1)$. The following, we will revisit the utility of this assumption following [19].

The main intuition behind the choice of initialization is that as information propagates through a neural network, either as forward flow of data or backward flow of gradients, there should be no systematic amplifications or attenuations. A convenient estimator for the 'amplitude' of the 'propagating signal' is its variance, which we now try to estimate. Let us begin with forward propagation.

Further, to fix our ideas, let us consider the flow of information as it passes from the ℓ^{th} to and through the $(\ell+1)^{\text{th}}$ layer. We have [13]

$$z_k^{(\ell+1)} = \sum_{j=1}^{n^{(\ell)}} w_{kj}^{(\ell+1)} z_j^{(\ell)} \,. \tag{7.3}$$

[12] Note that the p^{th} weights connect the $p-1$ layer to the p layer. If there are L hidden layers, p runs from 1 to $L+1$, where $L+1$ is the output layer.

[13] These results were derived for a linear network or a zero centred activation function in the vicinity of its zero, assuming $f'(0) = 1$. To fix our ideas, we can take the activation function to be tanh.

The Variance in $z^{(\ell+1)}$ results from fluctuations in $z^{(\ell)}$ and $w^{(\ell+1)}$. In the following we will take them to be formally independent variables. Firstly, note from the definition of covariance that

$$\mathrm{E}\left(z_j^{(\ell)}\right) = \mathrm{E}\left(w_{kj}^{(\ell)} z_j^{(\ell-1)}\right) = \mathrm{Cov}\left(w_{kj}^{(\ell)}, z_j^{(\ell-1)}\right) + \mathrm{E}\left(w_{kj}^{(\ell)}\right) * \mathrm{E}\left(z_j^{(\ell-1)}\right) . \tag{7.4}$$

Since $\mathrm{Cov}\left(w_{kj}^{(\ell)} z_j^{(\ell-1)}\right) = 0$ as they are independent and $\mathrm{E}\left(w_{kj}^{(\ell)}\right) = 0$, we must also have

$$\mathrm{E}\left(z_j^{(\ell)}\right) = 0 . \tag{7.5}$$

Then

$$\mathrm{Var}\left(z_k^{(\ell)}\right) = \mathrm{Var}\left(\sum_{j=1}^{n^{(\ell-1)}} w_{kj}^{(\ell)} z_j^{(\ell-1)}\right) = \sum_{j=1}^{n^{(\ell-1)}} \mathrm{Var}\left(w_{kj}^{(\ell)} z_j^{(\ell-1)}\right) , \tag{7.6}$$

where we used that if X and Y are independent then $\mathrm{Var}(X+Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$. Next, we use the result

$$\mathrm{Var}(XY) = \mathrm{E}(X)^2\,\mathrm{Var}(Y) + \mathrm{E}(Y)^2\,\mathrm{Var}(X) + \mathrm{Var}(X)\,\mathrm{Var}(Y) , \tag{7.7}$$

along with $\mathrm{E}\left(w^{(\ell)}\right) = 0$, and $\mathrm{E}\left(z^{(\ell-1)}\right) = 0$, [14] to conclude that

$$\mathrm{Var}\left(w_{kj}^{(\ell)} z_j^{(\ell-1)}\right) = \mathrm{Var}\left(w_{kj}^{(\ell)}\right) \mathrm{Var}\left(z_j^{(\ell-1)}\right) . \tag{7.8}$$

Then (7.6) yields

$$\mathrm{Var}\left(z_k^{(\ell)}\right) = \mathrm{Var}\left(w_{kj}^{(\ell)}\right) \sum_{j=1}^{n^{(\ell-1)}} \mathrm{Var}\left(z_j^{(\ell-1)}\right) . \tag{7.9}$$

Requiring that $\mathrm{Var}\left(z_k^{(\ell)}\right) = \mathrm{Var}\left(z_j^{(\ell-1)}\right)$ then yields

$$\mathrm{Var}\left(w_{kj}^{(\ell)}\right) = \frac{1}{n^{(\ell-1)}} . \tag{7.10}$$

Next, we need to study the evolution of variance as errors back propagate. Under our assumptions of the activation function we have

$$\delta_j^{(\ell)} = \sum_{k=1}^{n^{(\ell+1)}} \delta_k^{(\ell+1)} w_{kj}^{(\ell+1)} . \tag{7.11}$$

We will now carry out the same analysis as for the forward propagation. Firstly, note that

$$\delta_j^{(L)} = \sum_{k=1}^{n^{(K)}} \delta_k^{(L+1)} w_{kj}^{(L+1)} . \tag{7.12}$$

[14]Recall that $\mathrm{E}\left(z_j^{(\ell)}\right) = 0$. This argument works for all layers except $\ell = 1$ which is recieves inputs $z^{(0)}$ from the input layer. We can either assume that the inputs $z^{(0)} = x$ are zero centred or drop the first layer from this derivation and *a posteriori* use the obtained result on the first layer as well.

Further, the $w_{kj}^{(L+1)}$ are sampled from a symmetric distribution centred about zero. These together imply that

$$\mathrm{E}\left(\delta_j^{(L)}\right) = 0\,. \tag{7.13}$$

This is the same procedure as the one used to show $\mathrm{E}\left(z_j^{(1)}\right) = 0$ in forward propagation. Next,

$$\mathrm{Var}\left(\delta_j^{(L)}\right) = \mathrm{Var}\left(\sum_{k=1}^{n^{(L+1)}} \delta_k^{(L+1)} w_{kj}^{(L+1)}\right) = \sum_{k=1}^{n^{(L+1)}} \mathrm{Var}\left(\delta_k^{(L+1)} w_{kj}^{(L+1)}\right)\,. \tag{7.14}$$

We again have

$$\begin{aligned}\mathrm{Var}\left(\delta_k^{(\ell+1)} w_{kj}^{(\ell+1)}\right) &= \mathrm{E}\left(\delta_k^{(L+1)}\right)^2 \mathrm{Var}\left(w_{kj}^{(L+1)}\right) + \mathrm{Var}\left(\delta_k^{(L+1)}\right) \mathrm{E}\left(w_{kj}^{(L+1)}\right)^2 \\ &\quad + \mathrm{Var}\left(\delta_k^{(L+1)}\right) \mathrm{Var}\left(w_{kj}^{(L+1)}\right) \\ &= \mathrm{Var}\left(\delta_k^{(L+1)}\right) \mathrm{Var}\left(w_{kj}^{(L+1)}\right)\,,\end{aligned} \tag{7.15}$$

where we used $\mathrm{E}\left(\delta_j^{(L)}\right) = 0$ and $\mathrm{E}\left(w_{kj}^{(L+1)}\right) = 0$. As a result,

$$\mathrm{Var}\left(\delta_j^{(L)}\right) = n^{(L+1)}\,\mathrm{Var}\left(\delta_k^{(L+1)}\right) \mathrm{Var}\left(w_{kj}^{(L+1)}\right)\,. \tag{7.16}$$

Requiring that derivatives do not attenuate or amplify *via* backpropagation is formulated as the requirement that $\mathrm{Var}\left(\delta_j^{(L)}\right) = \mathrm{Var}\left(\delta_k^{(\ell+1)}\right)$. Hence

$$\mathrm{Var}\left(w_{kj}^{(L+1)}\right) = \frac{1}{n^{(L+1)}}\,. \tag{7.17}$$

Using the same set of arguments, now for the errors in the ℓ^{th} layer, yields

$$\mathrm{Var}\left(w_{kj}^{(\ell+1)}\right) = \frac{1}{n^{(\ell+1)}} \qquad \text{i.e.} \qquad \mathrm{Var}\left(w_{kj}^{(\ell)}\right) = \frac{1}{n^{(\ell)}}\,. \tag{7.18}$$

Equations (7.10) and (7.18) s are impossible to satisfy simultaneously. A reasonable compromise is to define

$$n_{av} = \frac{n_{in} + n_{out}}{2}\,, \qquad \text{and} \qquad w_{kj}^{(\ell)} \sim \mathcal{N}\left(0, \frac{1}{n_{av}^{(\ell)}}\right)\,. \tag{7.19}$$

This choice of initialization is known as *Xavier* initialization or *Glorot* initialization [19]. This is by no means a unique choice, several variations exist in the literature, different choices work better for different activation functions. For example, the preferred initialization with relu activation, which we shall come to in a moment, is *He initialization* [12]. However they are all united in that they are centred about 0 and have their variance suppressed by $\frac{1}{n}$, where n is determined in terms of the number of neurons in each layer of the DNN.

7.1.2 Non-Saturating Activation Functions

The second input to mitigate the vanishing gradients problem is the choice of activation function. So far we have at least implicitly chosen the activation function to be sigmoid throughout. This has been partly from intertia, continuing a choice made during Logistic Regression and while designing MLPs, and from intuition from biology, as biological neurons are believed to be sigmoid activated. [15] However, at least in principle we are free to choose other activation functions (typically non-linear, non-polynomial).

reLU: This is the rectified Linear Unit, whose functional form is

$$\texttt{relU}\left(z\right) = \max\left(0, z\right) \ . \tag{7.20}$$

This is almost a *de facto* standard choice when designing neural networks. This is a non-saturating function and clearly the derivative doesn't vanish for any positive z. However, the derivative is zero for negative z, and there is a possibility that the neuron may effectively 'die', outputting zero constantly, and its weights never being updated. It is possible to mitigate this problem both by altering the neural network architecture and by considering deformations of the relU activation.

Leaky reLU and variants: The leaky reLU is a deformation of the above reLU, and its functional form is

$$\texttt{leaky}\,\texttt{ReLU}\left(z\right) = \max\left(\alpha z, z\right) \ , \qquad 0 < \alpha < 1 \, , \tag{7.21}$$

i.e. the output at $z < 0$ is not zero, but 'leaks'. The parameter α may be fixed by hand, in which case it is a hyperparameter. Often, large values of *alpha* give good results. The `Keras` default is 0.3. Alternately, the randomized leaky ReLU (`RreLU`) scheme allows α to be sampled randomly during training and fixed to a mean value during testing. Finally, we have parametric leaky reLU or `PReLU` where α is a parameter to be learned by backprop.

Exponential Linear Unit: The exponential linear unit or `ELU` is a continuous and differentiable activation function given by

$$\texttt{ELU}\left(z\right) = \begin{cases} \alpha\left(e^z - 1\right) \ , & z \leq 0 \\ z \, , & z > 0 \end{cases} \ . \tag{7.22}$$

The reLU, leaky reLU and ELU activation functions are plotted in Figure 12 along with the sigmoid and tanh for reference. Their corresponding derivatives are plotten in Figure 13. We see that the derivatives of the sigmoid function are much smaller than all other activation functions.

7.2 Zero Centering Activations

Another reason why the sigmoid may not necessarily be an optimal choice for activation functions is because it is strictly positive on its domain. Consider a layer ℓ of sigmoid activated neurons. The gradients in this layer are

$$\frac{\partial E_\alpha}{\partial w^\ell_{jk}} = \delta^{(\ell)}_{j;\alpha} z^{(\ell-1)}_{k;\alpha} \ . \tag{7.23}$$

[15]Carrying over intuition from biology when designing ANNs can come with its pros and cons. In this example, it seems to hinder the design of effective ANNs more than anything. However, it is also known that biological neurons seldom, if ever, reach saturation. Maybe it is a question of which intuition should be carried over from biology, more than whether or not one should do so.

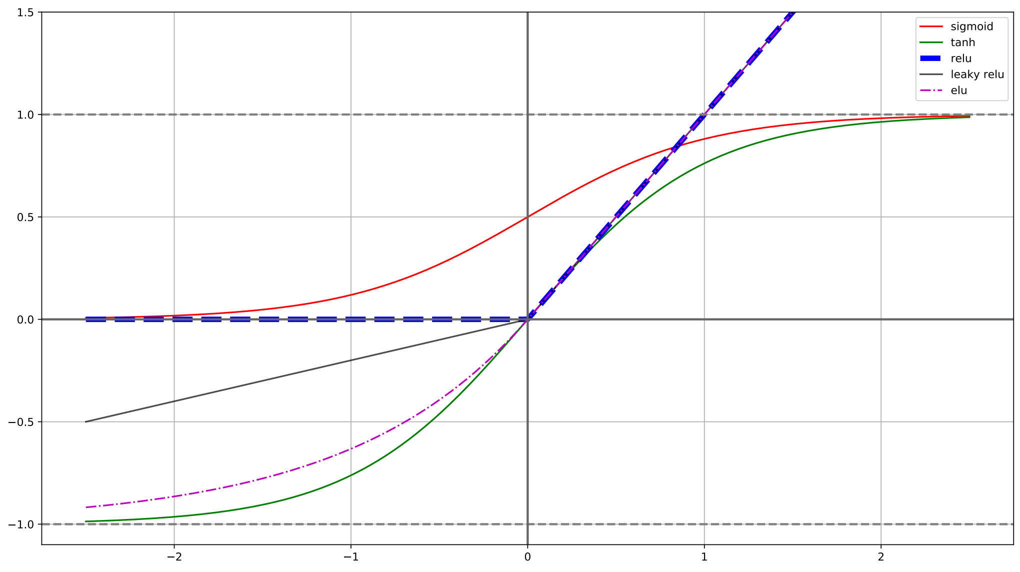


Figure 12: Common activation functions in Deep Learning. The sigmoid and tanh activations saturate while the others do not. All activations except sigmoid are zero centred.

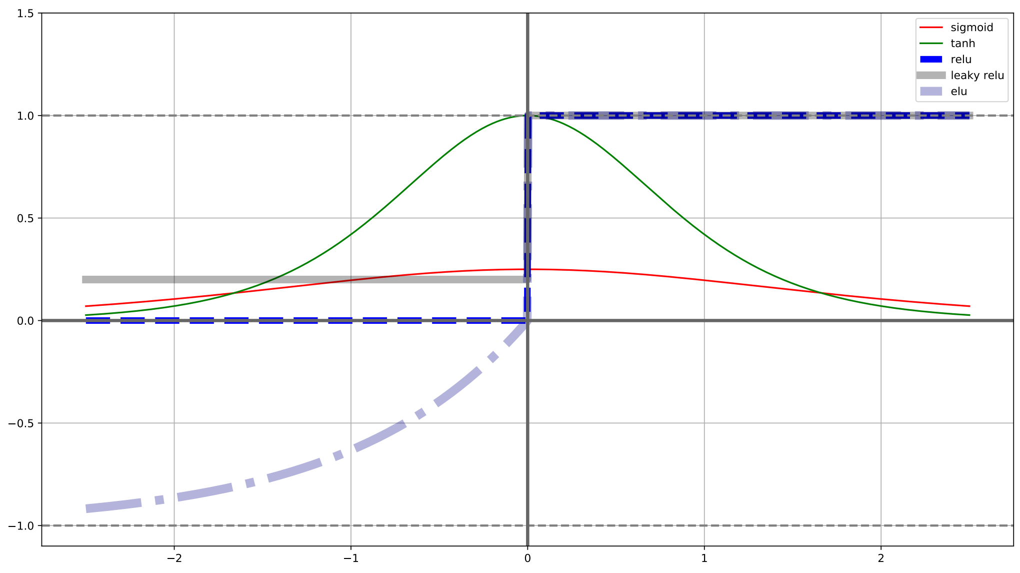


Figure 13: Derivatives of activation functions in Deep Learning. The derivatives of sigmoid and tanh activations saturate to zero while the others do not, except relu for negative arguments.

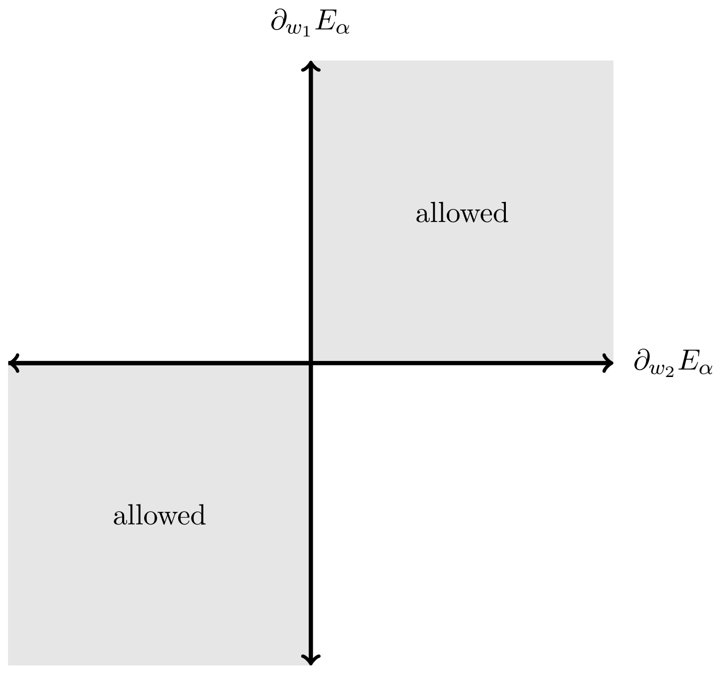


Figure 14: Allowed regions in gradient space, visualized for $w_{j1}^{(\ell)} \equiv w_1$ and $w_{j2}^{(\ell)} \equiv w_2$.

The term $\delta_{j;\alpha}^{(\ell)}$ is the same for all weights associated to the j^{th} neuron. Next, $z_{k;\alpha}^{(\ell-1)}$ is positive for all k. Hence, the gradients $\frac{\partial E_\alpha}{\partial w_{jk}^{\ell}}$ all have the same sign for all values of k given a fixed j. Either they're all positive or all negative. This severely reduces the space in which these gradients take values. This restriction is plotted for the case of two weights $w_{j1}^{(\ell)}$ and $w_{j2}^{(\ell)}$ in Figure 14. We see that only 2 of the possible 4 quadrants are accessible. If k can take 3 values then only 2 of the possible 8 quadrants are allowed. Similarly, the allowed region when k takes n values falls as 2^{1-n}. Thus, very quickly we find that only a tiny region of gradient space is allowed to us. While the restriction that all gradients carry the same sign is limited to a single neuron in a layer – in general different neurons in a layer do have different signs for gradients – and can also be circumvented by summing over a minibatch α, the general lesson that sigmoid activated neurons tend to hinder learning does hold true in practice.

From this we conclude that zero centred activations like tanh, which can equally well take negative values, are likely to be favoured over activations like sigmoid. The above arguments also hold on setting $\ell = 0$. This indicates that it is likely good practice to zero centre the inputs as well.

As a final note, we mention that many of the activations popular in deep learning are *not* zero centred. However, it is possible to implement zero centering by hand by means of a process called Batch Normalization, which is described below.

7.3 Batch Normalization

The choices we have made (Xavier initialization, non-saturating activation functions) help mitigate the vanishing gradients problem near the beginning of training, where we are still appreciably close to the initialization point. However, they do not guarantee that the problem doesn't return during training itself. Another strategy has proven to be quite useful in that regard. It is known as *Batch*

Normalization [20]. The intuitive idea is that at every training iteration we normalize the input to a given layer to a mean and variance which are learnt by backpropagation. This is in fact an extremely powerful technique and in fact, despite our prior floccinaucinihilipilification of the sigmoid and tanh neurons, incorporating Batch Normalization turns out to be enough to make a sigmoid or tanh activated neural network learn quite well in practice.

Here is the strategy. Consider a minibatch during training with data, and let the inputs to a given layer be $\vec{x}^{(i)}$, where $i = 1, 2, \ldots, m_B$ where m_B is the minibatch size. We compute

$$\begin{aligned} \vec{\mu}_b &= \frac{1}{m_B}\sum_{i=1}^{m_B} \vec{x}^{(i)}\,, \qquad & \vec{\sigma}^2_B &= \frac{1}{m_B}\sum_{i=1}^{m_B}\left(\vec{x}^{(i)} - \vec{\mu}_b\right)^2\,, \\ \hat{\vec{x}}^{(i)} &= \frac{\vec{x}^{(i)} - \vec{\mu}_b}{\sqrt{\vec{\sigma}^2_B + \epsilon}}\,, \qquad & \vec{z}^{(i)} &= \vec{\gamma} * \hat{\vec{x}} + \vec{\beta}\,. \end{aligned} \tag{7.24}$$

Then z is passed through the activation function. Here $\vec{\mu}_B$ and $\vec{\sigma}^2_B$ are recorded as moving averages during training, and then deployed as frozen numbers during testing. One may also do *Layer Normalization* where we normalize across all inputs $\vec{x}$ in a layer for a given training instance rather than across a minibatch, though we do not do so here.

7.4 Choosing the Learning Rate

The learning rate is a significant hyperparameter which affects the training of a neural network. Too low a value results in very slow convergence as the gradient descent takes very small steps in weight space. Too high a value on the other hand would cause the training to diverge. There are several strategies for choosing an 'optimal' learning rate. We especially emphasize two that were provided in [21].

Optimal Learning Rate The intuition behind this prescription is that the learning rate should be as large as possible, without becoming 'too large', i.e. causing the training to diverge. The following prescription is proposed.

1. Start with a very small learning rate, say $\eta = 10^{-6}$ or so.
2. At every training iteration (i.e. gradient descent step), increase the learning rate by a very small multiplicative amount.
3. Monitor the loss function. At a critical value η_{crit} the loss function starts to diverge as we carry out training iterations.

Choose the optimal value η_{max} to be one order of magnitude lower than this critical value [16]. The above is shown in Figure 15 where the curve was taken from an actual training instance.

[16] The one order of magnitude lower prescription comes from safety, as well as the fact that in practice most deep learning software do not record the loss function itself, but its moving average. The divergence that starts to appear in the loss function takes a bit of time to reflect in the moving average, hence we don't pick eta_{crit} itself.

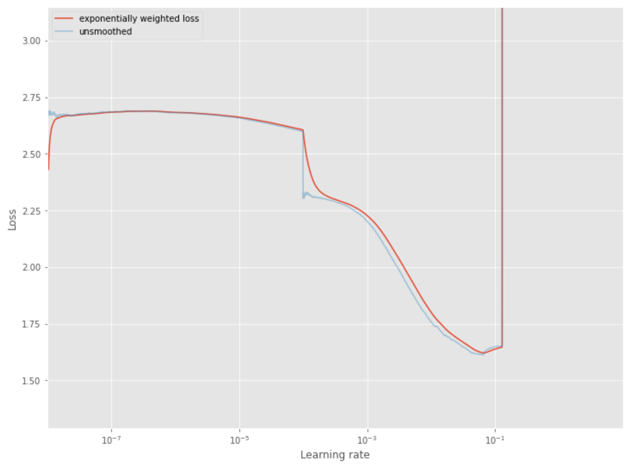


Figure 15: Evolution of the Loss function as the learning rate is increased every iteration.

One-Cycle or n-Cycle Scheduling We have obtained η_{max} above. Now define $\eta_{min} = \frac{\eta_{max}}{10}$ and $\eta_0 \approx 10^{-6}$. We then propose the following policy, shown in Figure. This dialing up from η_{min} to η_{max} and then down to η_{min} is called one cycle.

The intuition behind this is that the 'truly optimal' learning rate is somewhere between η_{max} and η_{min}. This policy allows us to spend 'most of the training' in the vicinity of this putative η_{opt}. Towards the end of the training we decay the learning rate to a small value η_0, allowing the model to tune its weights and biases more finely. This strategy extends to n cycles quite straightforwardly.

Other strategies to schedule the learning rate are

1. Power Scheduling
$$\eta\left(t\right) = \frac{\eta_0}{\left(1+\frac{t}{s}\right)^c}\,, \tag{7.25}$$
where we have to determine η_0, s, and c.

2. Exponential Scheduling
$$\eta\left(t\right) = \eta_0\left(0.1\right)^{t/s}\,, \tag{7.26}$$
where again η_0, s, are undetermined.

3. Piecewise Constant Scheduling
$$\eta\left(t\right) = \begin{cases} \eta_1, & 0 \le t < t_1\,, \\ \eta_2, & t_1 \le t < t_2\,, \\ & \vdots \end{cases} . \tag{7.27}$$
Typically only 2-3 steps are necessary. Also $\eta_1 > \eta_2 > \ldots$. The step lengths and the learning rates η_i have to be determined experimentally.

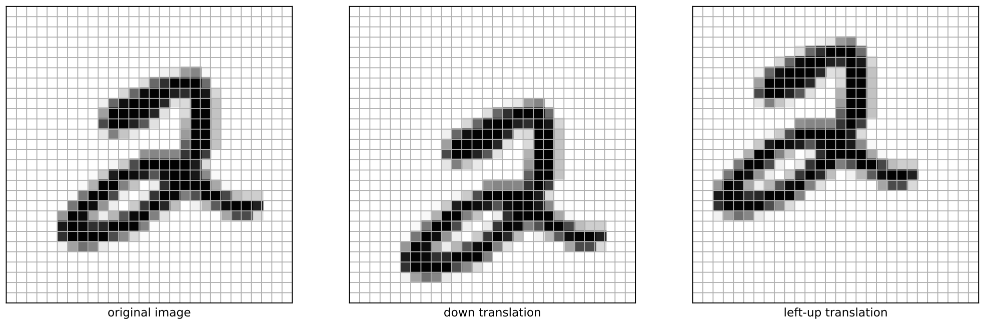


Figure 16: Translational invariance of images. These three images contain the same information, *viz.* the digit '2', irrespective of where the corresponding shape appears in the image. Note that if we were instead training a neural network to determine the location of the digit inside the image, the images would be treated as being translation *covariant* or equivariant since the expected output–the location of the '2'– would change by small amounts on changing the input slightly as above.

8 Convolutional Neural Networks

So far we have considered fully connected neural networks where neurons from a given layer are connected to all the neurons in the preceeding and succeeding layers. Such neural networks are an important test-bed for many ideas underlying neural network design and training. Further, fully connected layers are typically an integral part of neural network design even for more sophisticated architectures such as those discussed in this section. However, we have not yet considered the properties of the data that we are trying to machine learn. As a concrete example, if one is developing a neural network to analyze images, it is only natural to ask if and how do properties of images play a role in the formulating the design of the neural network being developed to analyze them [17].

There are two independent ways of motivating the design of convolutional networks, and it is particularly satisfying how such disparate points of view feed in to ultimately taking us towards the same goal. Firstly, from a biological point of view, one may take inspiration from the functioning of the human visual cortex. In particular, neurons in the visual cortex tend to have a small *local receptive field*, i.e. they only respond to stimuli in a certain part of the field of vision. Receptive fields of different neurons overlap, and together they span the whole field of vision. Next, some neurons react to only some particular low level features in the image and other neurons to other features even though they may be scanning the same receptive field. For example, some may be triggered by horizontal lines while others may be triggered by vertical lines. Other level neurons combine low level patters and synthesize them into more complex features. This system of synthesizing low level features into high level ones allows us to spot extremely rich visual patterns. Such inputs from the

[17] As an interesting parallel to the present discussion, note the existence of the *no free lunch theorem* [22]. This states that the performance of every machine learning algorithm averaged across all possible tasks is the same. It appears then that there is no 'best' machine learning algorithm simply because all machine learning algorithms do better on some tasks and worse on others. However, it is still meaningful to ask if some machine learning algorithms are better than others over *fixed tasks* such as image classification. The discussion in the present section may in some sense be regarded as an exemplification of such an approach.

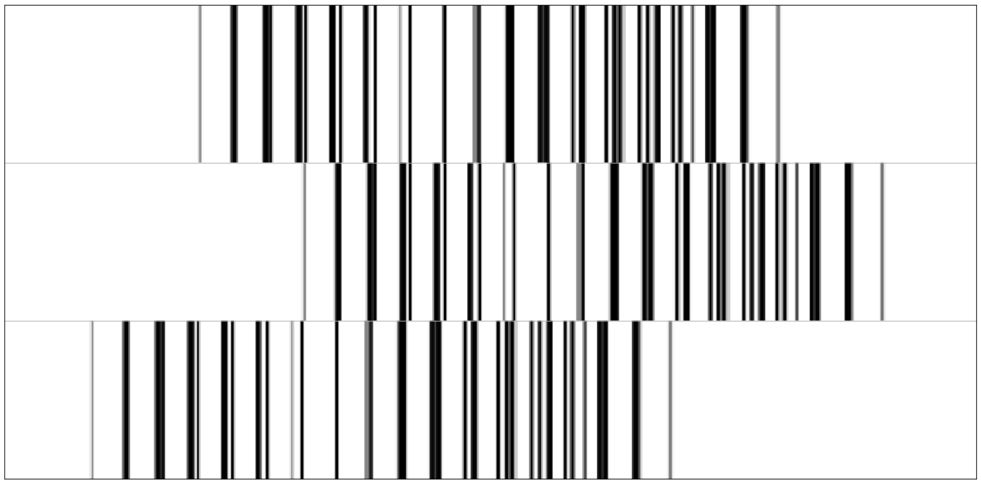


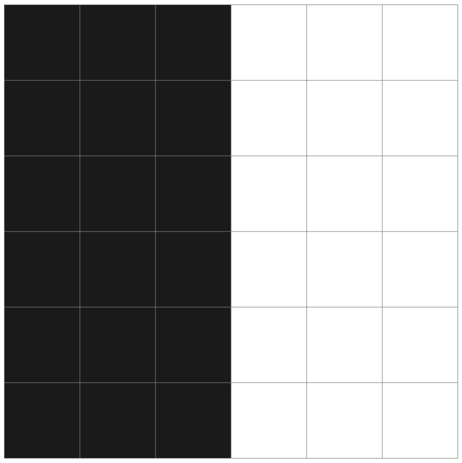
Figure 17: The images of Figure 16, left to right, unrolled into vectors and juxtaposed with each other, top to bottom.

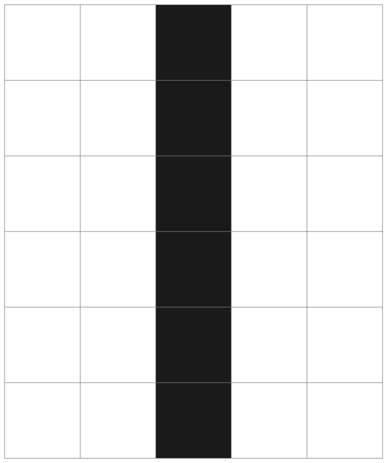
biological visual cortex were in fact key to the development of *convolutional neural networks* [23].

Somewhat complementarily, one may adopt a point of view more often associated with mathematical physics and ask what symmetries are our data likely to have. Then, these symmetries can be incorporated into the network design. For instance, images have a translational invariance which is easily seen from the Figure 16. However, this translational invariance is hard for fully connected networks to learn. To visualize this, consider 'unrolling' an image into a one dimensional column vector which is then to be fed into a MLP as in the previous sections. The unrolled vectors corresponding to the images in Figure 16 are shown juxtaposed with each other in Figure 17. We see from this figure that the local correlations visible in the grids of Figure 16 may get hidden when the image is unrolled. Some pixels which were previously close to each other get mapped far away on unrolling. Further, when fed into an MLP, an entirely different set of neurons will have to learn these correlations even when the image is slightly translated. These factors together make it very difficult for an MLP to learn complex image data.

Hence, we can formulate two assumptions about image data which will inform the design of the neural networks built for studying them. Firstly, that there there exist local patterns that repeat across the image. Secondly, that these patterns can be extracted and synthesized into still more complex patterns, much as the neurons in the visual cortex do.

Images as Tensors : Images are stored as (n_h, n_w, n_{ch}) tensors, where where the subscripts denote 'height', 'width' and 'channel'. For a grayscale image, $n_{ch} = 1$, so it effectively reduces to a two dimensional tensor or matrix of shape (n_h, n_w) where the matrix entry is the pixel intensity at the corresponding location. More generally, one could have a color image, for which $n_{ch} = 3$.


(a) Example image on a 6×6 grid consisting of a single vertical edge.


(b) Vertical edge extracted from this example image.

This typically stores pixel intensities for the three *additive primary colors*, Red, Green and Blue [18]. Further, intermediate layers in convolutional neural networks will have inputs and outputs which are three dimensional tensors where n_{ch} can be arbitrarily large.

8.1 A Simple Edge Detection Problem

Patterns can be extracted from studying gradations in the image. As an example, consider the 6×6 image of Figure 18a containing a single vertical edge, and its matrix representation given in (8.1).

$$\begin{pmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{pmatrix} \tag{8.1}$$

Intuitively it is clear that the edge is located where the pixel intensities, i.e. matrix entries, change the most sharply. We can extract the edge by the following simple prescription. We transform the image by taking the difference of every matrix entry with its right-adjacent one. We then get the matrix (8.2), visualized in Figure 18b.

$$\begin{pmatrix} 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 10 & 0 & 0 \end{pmatrix} \tag{8.2}$$

This was a very simple example where the feature was trivial to extract. However, in general one might hope to construct features by taking weighted local linear combinations of matrix entries as a

[18]Other encodings of color images are also possible, e.g. the *hue saturation value* format, which also results in $n_h \times n_w \times 3$ tensors.

natural generalization of the above computation. This is accomplished by *filters* and *convolutional layers*, to which we now turn.

8.2 Convolutional Layers

To construct a convolutional layer we begin by considering two $n \times m$ matrices A and B.

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nm} \end{pmatrix}. \tag{8.3}$$

Then the *Fröbenius inner product* between them is defined as (A, B) where

$$(A, B)_F = \sum_{ij} A_{ij} B_{ij} = \sum_{entries} \begin{pmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1m}b_{1m} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2m}b_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1}b_{n1} & a_{n2}b_{n2} & \dots & a_{nm}b_{nm} \end{pmatrix} = \mathrm{Tr}\left(A \cdot B^t\right). \tag{8.4}$$

In other words, the Fröbenius inner product is just the sum of entries of the Hadamard product. Another way of realizing this product is by unrolling the matrices into vectors. The Fröbenius inner product is then just the usual inner product over these vectors as seen here for 2×2 matrices.

$$\begin{aligned} \left(\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \right)_{\mathcal{F}} &= \begin{pmatrix} a_{11} & a_{12} & a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} \\ b_{12} \\ b_{21} \\ b_{22} \end{pmatrix} \\ &= a_{11}b_{11} + a_{12}b_{12} + a_{21}b_{21} + a_{22}b_{22}. \end{aligned} \tag{8.5}$$

If we have 3d tensors, say A_{ijk} and B_{ijk} of shape (n_h, n_w, n_{ch}), where k is the index along the channel direction then define the n_{ch} matrices indexed by k as

$$A_{ij}^{(k)} = A_{ijk}, \qquad B_{ij}^{(k)} = B_{ijk}. \tag{8.6}$$

Then

$$(A, B)_{\mathcal{F}} = \sum_{k=1}^{n_{ch}} = \left(A^{(k)}, B^{(k)}\right)_{\mathcal{F}}. \tag{8.7}$$

The above expressions make it manifest that taking the Fröbenius inner product of A with B is equivalent to taking a linear combination of entries of A where the coefficients are the entries of B. This is precisely the structure needed for extracting features from images, where A plays the role of the image, and B plays the role of the feature extractor.

We now define the convolution operation which we shall use to extract features. Recall that the intuitive prescription is to extract features by taking local linear combinations of image pixels. We can take linear combinations by the Fröbenius inner product. We define local regions by defining submatrices. In particular, we start by defining a *filter* F of size $f \times f$. Here f is called the *kernel* *size*. Then, given a matrix A, define $a^{(ij;f)}$, the $f \times f$ submatrix of A starting with the ij element on the top left. For example

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} , \quad a^{(21;2)} = \begin{pmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} , \quad a^{(12;3)} = \begin{pmatrix} a_{12} & a_{13} & a_{14} \\ a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} \end{pmatrix} . \tag{8.8}$$

Then, the *convolution* of A with F is defined as the matrix $\tilde{A}$ whose ij elements are given by

$$\tilde{A} \equiv A * F , \qquad \tilde{A}_{ij} = \left(a^{(ij;f)}, F \right)_{\mathcal{F}} , \tag{8.9}$$

where the $*$ in this section should not be confused with the Hadamard product. This convolution operation is visualized in Fig 20. It is also possible to generalize the above construction to *strided convolutions*. Instead of allowing i, j in (8.9) to take all values from 1 to $n_h - f + 1$ and $n_w - f + 1$ in steps of 1, we now take steps of size s. This implies we will restrict i and j to values of the form $\hat{\imath} \times s + 1$ and $\hat{\jmath} \times s + 1$ where $\hat{\imath}$ and $\hat{\jmath}$ start from 0.

It is also straightforward to generalize this to the three-dimensional case where the image also has a depth, i.e. is a tensor A of shape (n_h, n_w, n_{ch}). For the moment let us assume implicitly that n_{ch} corresponds to the color of the input image. We will shortly see that convolutions in three dimensions are indispensible even while working with grayscale images. Intuitively, we now need to take local linear combinations of the input image, along the depth as well. We start by taking the filter F to now be three dimensional and of shape (f, f, n_{ch}). Next we define the n_{ch}-plet of two dimensional tensors $f^{(k)}$ and $A^{(k)}$ as per (8.6). Then, the convolution of A with F is now

$$\bar{A} \equiv A * F = \sum_{k=1}^{n_{ch}} A^{(k)} * F^{(k)} , \tag{8.10}$$

with the $*$ on the right given by the two dimensional convolution (8.9). Another –equivalent– way of thinking about this operation is to view it as the replacement of the Fröbenius inner product (8.4) that appears in (8.9) by its three-dimensional extension (8.7).

Let us formalize this in the context of the vertical edge detection problem above. Here we are given a grayscale image, of shape $(n_h, n_w, 1)$ or equivalently, (n_h, n_w). We will therefore ignore the channel degree of freedom. Define the 3×3 filter

$$F = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} , \tag{8.11}$$

and carry out the above convolution (8.9) on the matrix (8.1). This will give

$$\begin{pmatrix} 0 & 10 & 10 & 0 \\ 0 & 10 & 10 & 0 \\ 0 & 10 & 10 & 0 \\ 0 & 10 & 10 & 0 \end{pmatrix} . \tag{8.12}$$

Further, as a more explicit illustration, we have shown the result of acting on an actual image with strong horizontal and vertical lines with the filter (8.11) and its horizontal counterpart in Figure 19.

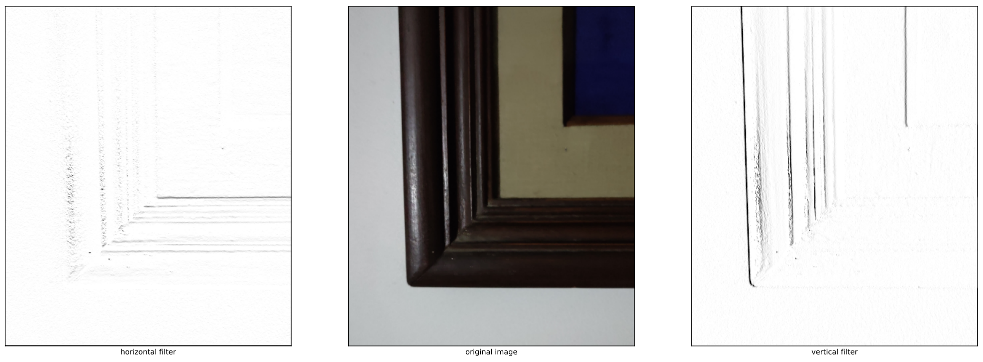


Figure 19: Acting on a test image with horizontal and vertical filters.

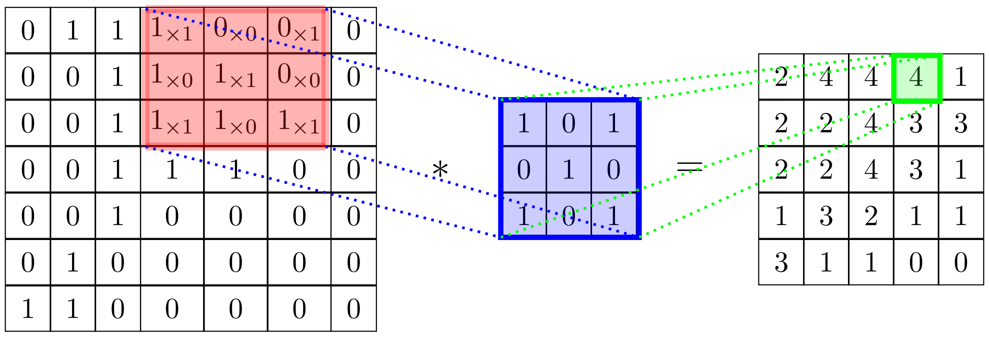


Figure 20: Visualizing the convolution operation. The filter scans the image with *stride* one.

The previous prescription of taking differences of adjacent points also falls in this framework. There the filter is not square, but a 2×1 matrix with entries $(1, -1)$. As we can readily see, the choice of filter for detecting a given feature is far from unique. As further examples, note two additional 3×3 filters that also detect vertical edges.

$$\text{Söbel filter}: \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}, \qquad \text{Scharr filter}: \begin{pmatrix} 3 & 0 & -3 \\ 10 & 0 & 10 \\ 3 & 0 & -3 \end{pmatrix}. \tag{8.13}$$

We can easily write down horizontal edge detecting filters by taking the transpose of the above filters. However, writing down filters that detect more complicated features is usually a harder task. Before deep learning, a significant amount of effort went into explicitly designing such filters for computer vision tasks [19]. In deep learning, we do not fix the form of the filter entries from

[19] An account of the general theory of constructing Söbel filters for diffent feature shapes is available on this

before and instead treat them as parameters which are to be learnt using backpropagation. This is much as how weighted linear combinations are constructed in the MLP architecture with the weights authorized to be learnt by backprop. Here the same kind of weighted combination is taken by these filters as well, only now over a local subset of the inputs. Also it is almost invariably necessary to take mutliple linear combinations, i.e. apply multiple filters, over local subsets of the inputs. Finally, note that the size of the filter is part of the neural network design.

Multiple Filters Recall that each filter detects a single feature, scanning across the input for its presence. To detect multiple features on the same input we need multiple filters. Let us define n_f filters F^c, each of size $f \times f \times n_{ch}$ and indexed by c, and scan across a (n_h, n_w, n_{ch})-shaped tensor called A using these filters. For every filter F^c we obtain an output $\bar{A}^c$ as per (8.10). Aggregating the n_f such outputs, the overall resultant is then a $n_h - f + 1 \times n_w - f + 1 \times n_f$ tensor. We can summarize this operation as

$$[n_h, n_w, n_c] * ([f, f, n_c])^{\otimes n_f} \to [n_h - f + 1, n_w - f + 1, n_f] \ . \tag{8.14}$$

Following this, we typically also apply a non-linear activation pointwise to every element of the output tensor. The set of all filters that scan over a given tensor, along with the application of the non-linearity, comprises a *convolutional layer*.

Padding: The convolution operations (8.9) and (8.10) have the following structure

$$[n_h \times n_w \times n_{ch}]_{\text{Image}} * [f \times f \times n_{ch}]_{\text{filter}} \to [(n_h - f + 1) \times (n_w - f + 1)] \ . \tag{8.15}$$

This has some downsides. For one, every time we pass a filter across the image, we shrink the image. Further, the pixels near the corner of the image are scanned relatively few times. Hence, it can happen that the information towards the edges can get thrown away. To mitigate these problems we introduce *padding*, i.e. we take the input image and introduce p rows and p columns of zeros on all sizes of the image. The resulting padded image is $n + 2p \times n + 2p$ in size and can now be convolved with a filter. The overall convolution operation is now

$$\begin{aligned}[n_h, n_w, n_{ch}]_{\text{Image}} &\xrightarrow[p]{\text{padding}} [n_h + 2p, n_w + 2p, n_{ch}]_{\text{Image}} \\ &\xrightarrow[\text{convolution}]{*(f,f)} [(n_h + 2p - f + 1)\,, (n_w + 2p - f + 1)] \ .\end{aligned} \tag{8.16}$$

A popular choice for p is

$$p = \frac{f-1}{2}\,, \tag{8.17}$$

so that the input and the output images have the same size. This is called *same padding*. Typically f is taken to be odd in which case p is an integer. Other choices of padding are also possible, and are used in practice as well. The generalization to multiple filters is apparent.

Parameter Count: To fix our ideas let us imagine that a convolutional layer consists of 10 filters which are $3 \times 3 \times 3$ each. The total number of parameters is

$$10 \times (3 \times 3 \times 3 + 1) = 280\,. \tag{8.18}$$

Wikipedia page.

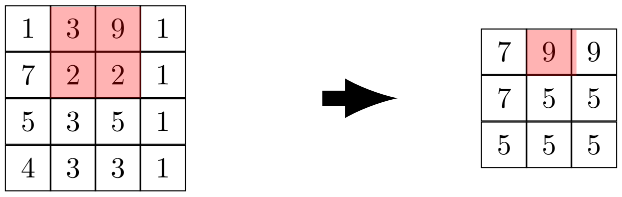


Figure 21: Illustration of Max Pooling. To get Average Pooling, take mean of numbers in the red shaded area instead of the max as done here.

The 1 is the bias term for each filter. Note that unlike fully connected layers, the parameter count does not scale with the sizes of the inputs to the layer and the layer outputs. Instead, it scales with the number of features and size of local field of vision, i.e. f, both of which are typically much smaller numbers.

At the same time one must be careful to remember that convolutional networks are very intensive on memory. From the above schematic in (8.14), we see that the output of the layer is n_f times the size of the input. With multiple layers and even a moderate number of filters in each layer the total memory required to even fit a single forward propagation of the neural network can become quite prohibitive. Pooling layers and 1×1 convolutions, also known as network-within-networks, mitigate these memory requirements. We now turn to these.

8.3 Pooling Layers

One of the most important The operation of a pooling layer is similar to that of a convolutional layer. Only instead of taking a linear combination of the image elements in the field of vision, we 'pool' all the image elements in the field of vision and take their average or maximum typically. The first option is called Average Pooling and the second, Max Pooling. The operation is illustrated in Figure 21 for Max Pooling. More formally, we start with an input tensor $Z^{(\ell)}$ of shape (n_h, n_w) on which we will apply a Pooling operation of kernel size $f \times f$. To do so, we define $Z^{(\ell)}_{(ij;f)}$, the $f \times f$ submatrix of $Z^{(\ell)}$ starting with the ij element on the top left. This was concretely illustrated in (8.8). Then, for Max Pooling, the output $Z^{(\ell+1)}$ is given by the matrix

$$Z^{(\ell+1)}_{ij} = \max\left(Z^{(\ell)}_{(ij;f)}\right) . \tag{8.19}$$

The case of Average Pooling is identical, the only difference being that instead of picking the maximum of elements in $Z^{(\ell)}_{(ij;f)}$, we pick their average. The case of the three dimensional tensors is identical, the depth direction is only a spectator. That is, we break up the input tensor into n_{ch} components $Z^{(\ell);k}$ indexed by k and apply the $2D$ Pooling operation for each component, then reassemble everything back into the three dimensional tensor $Z^{(\ell+1)}$, now of shape $(n_h - f + 1, n_w - f + 1, n_{ch})$.

As is apparent, although pooling mitigates the memory requirements of training a convnet, it is quite a destructive operation where a large fraction of the image information is wiped out. Nonetheless pooling is not just a method of mitigating memory requirements and can be actively advantageous to include in the network design.

Firstly, one effect of pooling is of *coarse-graining* the input tensor, i.e., reducing the size while trying to retain as much of the information in the tensor as possible. The higher level filters scanning the resulting tensor also have a much larger field of vision, without increasing the filter size.

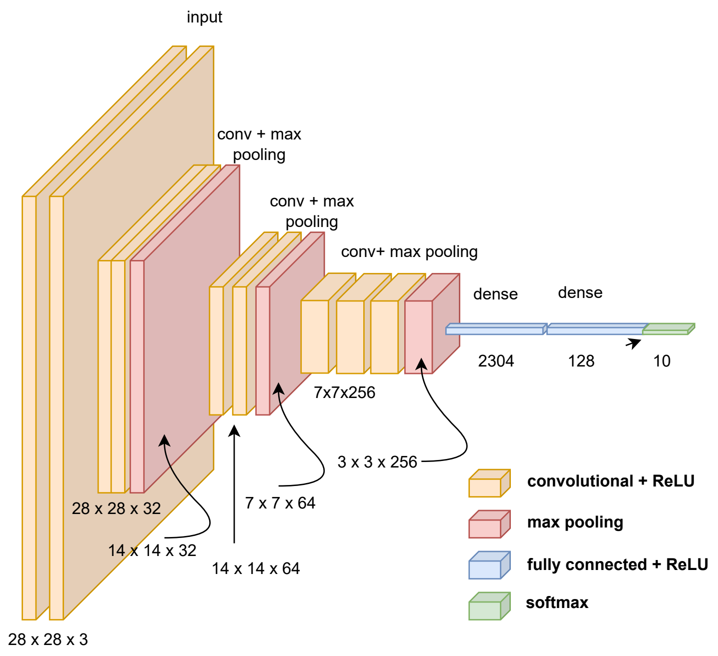


Figure 22: An elementary convnet. The convolutional layers have ‘same’ padding and the Max Pooling layers have a kernel size of 2×2 with stride 2.

Secondly, it is also a means of encoding symmetry within the neural network design. For example, in object classification we may simply wish to determine whether or not an image contains a particular object, rather than knowing precisely where in the image the object is. Pooling can help in extracting robust features associated with this object.

Finally, one could also do pooling along the third axis, which would reduce the output to $n_h \times n_w \times 1$. These are all very destructive operations, but again useful in helping the convnet learn notions of invariance, going beyond the translational invariance of $2D$ Pooling. Suppose we wish to have a rotational invariance in the image, so that horizontal, vertical or skew edges are on an equal footing, and a layer in the convnet consists of 3 filters that detect these edges. Then, max pooling along the depth direction would tend to make the output the same regardless of which filter is actually triggered.

8.4 Putting it all together: A basic ConvNet

The above elements of convolutional layers, along with pooling and fully connected layers of the previous sections are foundational to many pathbreaking neural networks such as the LeNet, AlexNet, and the VGG series. Other network designs such as the Inception or the Residual Nets are specific design changes to this overall paradigm. We will now outline the design of an elementary convolutional neural network using these elements. We start with an input image of shape (n_h, n_w, n_{ch}) and act on it with a *convolutional layer* consisting of n_f filters of kernel size f. For definiteness we will also take same padding. This *aggregation operation* is followed by the *coarsening operation* of Pooling. Finally, the output of these operations is fed to the next convolutional layer and the cycle repeats a few times. Typically an image has a few low level features, e.g. horizontal, vertical and skew lines, and many high level features, e.g. leaves, petals, eyes, mouths. It is therefore usually the case that the number of filters increases as we move deeper into the network. The output of the convolutional layers is then unrolled *a la* (8.5) and then fed to a few dense layers. Finally, the output is collected. Just as the role of the hidden layers in the fully connected MLP was to generate features on which the output layer can do Logistic Regression (say), the role of the convolutional layers can be thought of as generating features through local correlations in the image on which the MLP can learn.

8.5 A Network within a Network

Typically in a single layer there can be a large number of features, e.g. 32, 64 or 128 is not uncommon. Hence, as we pass through multiple convolutional layers the number of channels can grow rapidly. We have already seen one strategy, pooling, to mitigate this growth. Another strategy to mitigate the growth in n_{ch} as we pass through convolutional layers is to act with a 1×1 convolution. The structure of this operation is

$$[n_h, n_w, n_{ch}] * [1, 1, n_{ch}]^{\otimes n_f} \to [n_h, n_w, n_f] \ . \tag{8.20}$$

Then, if $n_f < n_c$ we have reduced n_c while preserving n_h and n_w. We now write out this operation more explicitly to understand its action. A $1 \times 1 \times n_{ch}$ filter is a tuple of numbers f_p where $p = 1, 2, \ldots, n_{ch}$. In the same vein, the elements of all such n_f filters can be viewed as a $n_f \times n_c$ matrix $\tilde{f}$, written as

$$\tilde{f} = \begin{pmatrix} f_{11} & f_{12} & \cdots & f_{1n_{ch}} \\ f_{21} & f_{22} & \cdots & f_{2n_{ch}} \\ \vdots & \vdots & \ddots & \vdots \\ f_{n_f 1} & f_{n_f 2} & \cdots & f_{n_f n_{ch}} \end{pmatrix}_{n_f \times n_{ch}} \ . \tag{8.21}$$

We denote its entries by f_{qp}. Consider the output Z^ℓ from layer ℓ of the convolutional network and let it be a tensor of shape $[n_h, n_w, n_{ch}]$. We will construct the output of the $\ell + 1^{\text{th}}$ layer. Define the elements $Z^{(\ell)}_{ijk}$ for fixed ij, i.e. along the channel direction of the input tensor. Suppressing the ij indices to write $Z^{(\ell)}_{ijk} \equiv z^{(\ell)}_k$, the output of acting with these filters is given by the tuple $a^{(\ell+1)}_q$ where

$$\begin{pmatrix} a^{(\ell+1)}_1 \\ a^{(\ell+1)}_2 \\ \vdots \\ a^{(\ell+1)}_{n_f} \end{pmatrix} = \begin{pmatrix} f_{11} & f_{12} & \cdots & f_{1n_{ch}} \\ f_{21} & f_{22} & \cdots & f_{2n_{ch}} \\ \vdots & \vdots & \ddots & \vdots \\ f_{n_f 1} & f_{n_f 2} & \cdots & f_{n_f n_{ch}} \end{pmatrix} \cdot \begin{pmatrix} z^{(\ell)}_1 \\ z^{(\ell)}_2 \\ \vdots \\ z^{(\ell)}_{n_{ch}} \end{pmatrix} \ . \tag{8.22}$$

Finally, to construct the output, we will apply a nonlinearity to $a^{(\ell+1)}$ and define

$$z^{(\ell+1)} = h^{[\ell]} \left(a^{(\ell+1)} \right) \ . \tag{8.23}$$

Doing this operation for all ij values yields the output tensor $Z^{(\ell+1)}$. Thus we see from equations (8.21), (8.22), (8.23), that the structure of the 1×1 convolutional layer is that of a Fully Connected layer acting along the channel direction only. This gives rise to the somewhat whimsical name of *network-in-network* for this layer.

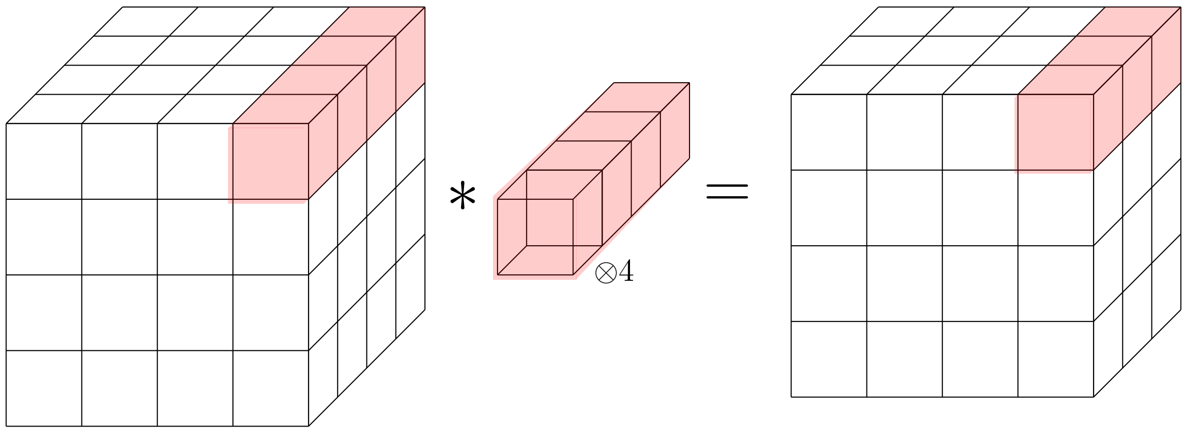


Figure 23: Network in a Network. $n_f = 3$ filters of shape $1 \times 1 \times 4$ convolved with $4 \times 4 \times 4$ shaped inputs produce an activation of shape $4 \times 4 \times 3$.

8.6 Convolutions are All You Need

We have seen that a 1×1 convolution acts as a fully connected network along the depth direction. Somewhat complementary to this is the idea of a *fully convolutional neural network* where we replace the dense layers in a convolutional network by convolutional layers [24]. To see how, let the output of the final convolutional layer be a tensor Z^ℓ of shape (n_h, n_w, n_{ch}). For definiteness we take the shape to be $(2, 2, 2)$. This gets unrolled into the vector

$$Z^\ell_{flat} = \begin{pmatrix} z_{111} & z_{121} & z_{211} & z_{221} & z_{112} & z_{122} & z_{212} & z_{222} \end{pmatrix}^T . \tag{8.24}$$

Then, the preactivation of the fully connected layer $\ell + 1$ of size $n_{\ell+1}$ is

$$\tilde{a}^{(\ell+1)}_m = \sum_{I \in \{(ijk)\}} W^{(\ell+1)}_{mI} \tilde{Z}^\ell_I , \tag{8.25}$$

where I runs over all distinct ordered triplets (ijk) and $m = 1, 2, \ldots, n_{\ell+1}$. Clearly we can also write this as

$$\tilde{a}^{(\ell+1)}_m = \sum_{ijk} W^{(\ell+1)}_{m;ijk} \tilde{Z}^\ell_{ijk} , \tag{8.26}$$

where the sums over i, j, k run independently from 1 to n_h, n_w and n_{ch} respectively.

Next, consider the action of a convolutional layer with $n_{\ell+1}$ filters and kernel size (n_h, n_w) on Z^ℓ. As per (8.10), the output should be a tensor of shape $(1, 1, n_{\ell+1})$. More explicitly, the output is

$$a^{\ell+1}_m = \sum_{ijk} F^{(m)}_{ijk} Z^{(\ell)}_{ijk} \tag{8.27}$$

Clearly on identifying $F^{(m)}_{ijk}$ with $W^{(\ell+1)}_{m;ijk}$ we see that the convolutional layer is computing the same sum as the dense layer without having to unroll the network. The action of the subsequent dense layers can be implemented by passing through 1×1 convolutions with the feature number given by the dimension of the dense layer being replaced. Thus, we can replace all the dense layers, by convolutional layers, which leads to a fully convolutional network. In practice this is very useful when we wish to train the network on images of one size but deploy it on images of different sizes.

9 Conclusions & Outlook

We have covered in these notes an overview of some common threads that underly machine learning methods, whether neural networks or otherwise, along with an introduction to some concrete methods. These were regression using Generalized Linear Models, Logistic Regression for classification, and Neural Networks. Inevitably, sins of omission exceed any services of commission and we have left out far more than we have covered. Decision Trees, Nearest Neighbour Classifiers, Support Vector Machines, Ensembling, Sequence Models, Generative Models come most readily to mind, along with unsupervised and semi-supervised learning. Concrete applications of these methods to problems in string theory and mathematical physics have also been omitted, motivated in part by our desire to emphasize the general ideas underlying these methods which are already suggestive of great universality in application. We refer the reader to for further reading and details.

Acknowledgements

Figures 10 and 11 were generated by minor changes to the Tikz codes by Izaak Neutelings available here and the Figure 22 was generated similarly starting from the diagrams.net files made by Kenneth Leung available here. We thank Gurmeet Singh for interesting discussions about role of universal approximation theorems in machine learning. Finally, these notes are only a very slightly expanded review of a short lecture series given at ICTS–TIFR Bengaluru and the Department of Physics, IIT Kanpur to graduate and masters students. We especially thank them for their patience and for several interesting discussions.

Appendix

A Bias/Variance Decomposition

We have so far provided a heuristic explanation of bias, variance and the bias/variance decomposition. We now turn to a more abstract discussion where can provide quantitative expressions for bias and variance which could provide more insight. This closely follows the treatment in [3]. As always, we seek to model data

$$\mathcal{D} = \{(x_\alpha, t_\alpha)\} \ . \tag{A.1}$$

Further, there is some underlying stochasticity in the data due to which we postulate that it is drawn from the joint probability distribution $p\,(x, t)$. [20] Given this framework, what is the best we can do in terms of predicting t given a value of x?

To answer this question, let us return to the discussion in Section 2, and let f be a function that predicts the value of t given a value of x. Further, let $\mathcal{F}$ be the set of all prediction functions. We also define the *loss* $L\,(t, f\,(x))$ which gives a measure of the mismatch between the actual and predicted values for a given x. Then the prediction function which is *optimal*, in the sense that it minimizes the expected loss

$$\mathrm{E}\,[t, f\,(x)] = \int L\,(t, f\,(x))\,\mathrm{d}p\,(x, t) \ , \tag{A.2}$$

[20] As remarked previously, in general $p\,(x, t)$ is prohibitively difficult to compute. However the explicit knowledge of the probability distribution is not required for the present discussion, only its existence is.

is denoted by $\tilde{f}$. Explicitly determining $\tilde{f}$ requires both the knowledge of the joint probability distribution $p(x,t)$ and the ability to scan over the whole $\mathcal{F}$. Nonetheless, we can formally evaluate it for the mean square loss (2.5)

$$L(t, f(x)) = (t - f(x))^2 \; . \tag{A.3}$$

using the calculus of variations. In particular, using

$$\mathbb{E}[L] = \iint (t - f(x))^2 \, p(x,t) \, dx \, dt \, , \tag{A.4}$$

$\tilde{f}$ is given by (A.4).

$$0 = \left. \frac{\delta \mathbb{E}[L]}{\delta f(x)} \right|_{f=\tilde{f}} = -\int \left(t - \tilde{f}(x) \right) p(x,t) \, dt \, . \tag{A.5}$$

This may be solved using

$$\int t p(x,t) \, dt = \int \tilde{f}(x) \, p(x,t) \, dt = \tilde{f}(x) \, p(x) \; . \tag{A.6}$$

Thus $\tilde{f}$ is given by

$$\tilde{f}(x) = \int t \frac{p(x,t)}{p(x)} dt = \int t \, p(t|x) \, dt \, . \tag{A.7}$$

This is the expectation value of t conditioned on x and is the optimal prediction function, called the *regression function*. There is another, more intuitive, sense in which $\tilde{f}$ is optimal. To see this, decompose

$$\begin{aligned} (f(x) - t)^2 &= \left(f - \tilde{f} + \tilde{f} - t \right)^2 \\ &= \left(f - \tilde{f} \right)^2 + 2 \left(f - \tilde{f} \right) \left(\tilde{f} - t \right) + \left(\tilde{f} - t \right)^2 \; . \end{aligned} \tag{A.8}$$

Now [21]

$$\iint \left(f(x) - \tilde{f}(x) \right) \left(\tilde{f} - t(x) \right) p(x,t) \, dx dt = 0 \, . \tag{A.11}$$

As a result, the expression for the expected loss is the following sum of squares.

$$\mathbb{E}[L] = \iint \left(f(x) - \tilde{f}(x) \right)^2 p(x,t) \, dx \, dt + \iint \left(t - \tilde{f}(x) \right)^2 p(x,t) \, dx \, dt \, . \tag{A.12}$$

In this, the second term cannot be changed by changing $f(x)$. It originates from the fact that t itself is a stochastic variable, and may be regarded as an intrinsic noise term. The first term vanishes

[21] To see this, note that

$$\begin{aligned} \iint \left(f(x) - \tilde{f}(x) \right) t \, p(x,t) \, dx dt &= \int dx \left(f(x) - \tilde{f}(x) \right) \left[\int dt \, t \, p(t|x) \right] p(x) \\ &= \int dx \left(f(x) - \tilde{f}(x) \right) \tilde{f}(x) \, p(x) \; . \end{aligned} \tag{A.9}$$

Hence,

$$\iint \left(f(x) - \tilde{f}(x) \right) \left(\tilde{f} - t(x) \right) p(x,t) \, dx dt = 0 \, . \tag{A.10}$$

when we choose $f(x)$ to be $\tilde{f}(x)$. Hence we have shown that (A.12), and by conjunction (A.4), are minimized for the regression function $\tilde{f}$. Further, any residual error in prediction is purely due to the stochasticity of the target variable itself and cannot be improved by better model selection.

Next, we turn to defining bias and variance in making predictions $y(x)$ from a parametric model $y(x;w)$. Also recall our previous discussion that at least on a heuristic level, models can fail to generalize well either because our model assumptions fail to capture the underlying regularities actually present in the data (i.e. *bias*) or the model is susceptible to learning peculiarities present in the data it is trained on, which are not present in the overall data (i.e. *variance*). This indicates that bias is associated to the mismatch between the predictions we expect by optimizing the parametric model $y(x;w)$ on some training data, while variance is associated to the spread in the predictions obtained as we vary the data on which $y(x;w)$ is trained on.

To estimate these quantities, we propose the following thought experiment. Suppose we are given not one but an ensemble of datasets $\{\mathcal{D}\}$. In practice, such an ensemble may be generated by drawing subsets from available data with or without replacement. We train our parametric model over each dataset in the ensemble to obtain a function $y(x;D)$ using which we compute and decompose the loss.

$$\begin{aligned}\left(y(x;\mathcal{D})-\tilde{f}(x)\right)^2 &= \left(y(x;\mathcal{D})-\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]+\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]-\tilde{f}(x)\right)^2\\ &= \left(y(x;\mathcal{D})-\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]\right)^2+\left(\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]-\tilde{f}(x)\right)^2\\ &+2\left(y(x;\mathcal{D})-\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]\right)\left(\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]-\tilde{f}(x)\right)\,.\end{aligned} \tag{A.13}$$

The expectation value of this loss over the ensemble of datasets is given by

$$\mathbb{E}_{\mathcal{D}}\left[\left\{y(x;\mathcal{D})-\tilde{f}(x)\right\}^2\right]=\left\{\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]-\tilde{f}(x)\right\}^2+\mathbb{E}_{\mathcal{D}}\left[\left\{y(x;\mathcal{D})-\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]\right\}^2\right]\,, \tag{A.14}$$

where we used

$$\mathbb{E}_{\mathcal{D}}\left[\left\{\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]-\tilde{f}(x)\right\}^2\right]=\left\{\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]-\tilde{f}(x)\right\}^2\,, \tag{A.15}$$

and

$$\begin{aligned}&\mathbb{E}_{\mathcal{D}}\left[\left\{\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]-\tilde{f}(x)\right\}\left\{y(x;\mathcal{D})-\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]\right\}\right]\\ &=\left\{\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]-\tilde{f}(x)\right\}\mathbb{E}_{\mathcal{D}}\left[\left\{y(x;\mathcal{D})-\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]\right\}\right]\\ &=\left\{\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]-\tilde{f}(x)\right\}\left\{\mathbb{E}_{\mathcal{D}}\left[y(x;\mathcal{D})\right]-\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]\right\}=0\,.\end{aligned} \tag{A.16}$$

Thus we see that the expected generalization error obtained from training $y(x,w)$ on finite datasets, evaluated in (A.14), is a sum of two terms.

$$\begin{aligned}\texttt{bias}\quad &:\quad \left\{\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]-\tilde{f}(x)\right\}^2\\ \texttt{variance}\quad &:\quad \mathbb{E}_{\mathcal{D}}\left[\left\{y(x;\mathcal{D})-\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]\right\}^2\right]\,.\end{aligned} \tag{A.17}$$

The *bias* term captures the extent to which the expected output differs from the desired regression function. The *variance* on the other hand is how much each function $y(x;\mathcal{D})$ differs from the average over datasets, and represents the sensitivity of $y(x)$ to the choice of dataset it is actually trained on. Quantitatively, it is the standard deviation about the mean prediction as we vary the training data.

This analysis was for a single point x. We now average as in (A.12) to obtain the expression

$$\text{generalization error} = \text{bias} + \text{variance} + \text{noise}\,, \tag{A.18}$$

$$\begin{aligned}
\texttt{bias} \quad &: \quad \iint \left\{\mathbb{E}_{\mathcal{D}}\left[y(x;D)\right] - \tilde{f}(x)\right\}^2 p(x,t)\,dx\,dt\,,\\
\texttt{variance} \quad &: \quad \iint \mathbb{E}_{\mathcal{D}}\left[\left\{y(x;\mathcal{D}) - \mathbb{E}_{\mathcal{D}}\left[y(x;D)\right]\right\}^2\right] p(x,t)\,dx\,dt\,,\\
\texttt{noise} \quad &: \quad \iint \left(t - \tilde{f}(x)\right)^2 p(x,t)\,dx\,dt\,.
\end{aligned} \tag{A.19}$$

As pointed out previously, though we would like to drive down both bias and variance to minimize the generalization error, in practice the two cannot be reduced independently of each other. Models which have low bias tend to have high variance and models with low variance tend to have high variance.

B A Bayesian Treatment of Regression

We now turn to a Bayesian treatment of generalized linear regressors. The Bayesian formulation of machine learning is quite distinct to what we have done so far, which may be thought of as frequentist in nature, where uncertainity in the model parameters arises from fluctuations in data. This is perhaps most vividly illustrated in our characterization of bias and variance. By this token, confidence in the determined values of model parameters should be estimated by repeated 'experiments' where the model is trained over multiple datasets. On the other hand, in a Bayesian picture, uncertainity in the model is due to our lack of knowledge about the parameters themselves. We start with prior assumptions which we keep updating in the face of evidence in accordance with Bayes theorem

$$p(a|b) = \frac{p(b|a)\,p(a)}{p(b)}\,. \tag{B.1}$$

Typically such Bayesian analyses are prohibitively difficult and almost certainly impossible to carry out analytically except in approximation schemes. However, for GLM regressors and with some assumptions regarding how the data is distributed, they can be carried out exactly. An engaging review of the history of the Bayes theorem is available here.

We return to our regression problem, adopting the notation of the previous sections, i.e. we assume we are given data $\mathcal{D}$ in the form of N input-output pairs $\{(x_n, t_n)\}$, on the basis of which we would like to predict t given a value of x. Again, we also assume that t arises from an underlying regularity expressible as an adaptive model $y(x,w)$ with some noise, i.e.

$$t = y(x;w) + \text{noise}\,. \tag{B.2}$$

We choose to model this by assuming that the observations t_n corresponding to x_n are independently drawn from the probability distribution functions

$$\mathcal{N}\left(t|y(x;w), \beta^{-1}\right)\,, \tag{B.3}$$

where

$$\mathcal{N}\left(t|\mu,\beta^{-1}\right)=\left(\frac{\beta}{2\pi}\right)^{\frac{1}{2}}\exp\left(-\frac{\beta}{2\pi}\left(t-\mu\right)^{2}\right)\ . \tag{B.4}$$

Here β is the inverse variance, otherwise known as the *precision*. It, along with the parameters w are undetermined thus far. We now turn to some methods for estimating the values of these parameters. We assume a generalized linear model ansatz

$$y\left(x,w\right)=\sum_{j=0}^{M}w_{j}\phi_{j}\left(x\right)\ . \tag{B.5}$$

Finally, we shall sometimes denote

$$\vec{x}=\begin{pmatrix}x_1, & x_2, & \dots, & x_N\end{pmatrix}\ ,\qquad\text{and}\qquad\vec{t}=\begin{pmatrix}t_1, & t_2, & \dots, & t_N\end{pmatrix}\ . \tag{B.6}$$

B.1 Maximum Likelihood Estimation

Given our assumptions about how t is distributed, the *likelihood* for observing data $\mathcal{D}$ is just

$$p\left(t|x,w,\beta\right)=\prod_{\alpha=1}^{N}\mathcal{N}\left(t_{\alpha}|w\cdot\phi\left(x_{\alpha}\right),\beta^{-1}\right)\ . \tag{B.7}$$

We postulate that the parameters w are such that the above likelihood function is maximized, i.e. w are such that probability of observing any data is maximized for $\mathcal{D}$, the dataset actually observed in practice.

Maximizing the likelihood is equivalent to minimizing the the *log-likelihood*, given by

$$\ln p\left(t|x,w,\beta\right)=\sum_{\alpha=1}^{N}\ln\mathcal{N}\left(t_{\alpha}|w\cdot\phi\left(x_{\alpha}\right),\beta^{-1}\right)\ . \tag{B.8}$$

Next, using (B.4),

$$\ln p\left(t|x,w,\beta\right)=\sum_{\alpha=1}^{N}\frac{1}{2}\ln\beta-\frac{1}{2}\ln 2\pi-\frac{\beta}{2}\left(t_{\alpha}-w\cdot\phi\left(x_{\alpha}\right)\right)^{2}\ . \tag{B.9}$$

As a result, minimizing the log-likelihood function is equivalent to minimizing the mean square error function (3.5) obtained previously from geometric intuition. We note however that assuming a different probability distribution and a different likelihood function would in principle alter the maximum likelihood criterion so it would then no longer match with the MSE criterion.

Returning to the minimization of the log-likelihood, the minimum in w is located at the minima of (3.5), given by

$$w_{ML}=w_{*}=\left(\Phi^{T}\Phi\right)^{-1}\Phi^{T}\,t\,, \tag{B.10}$$

with Φ being the $N\times M$ *design matrix* with matrix entries

$$\Phi_{\alpha j}=\phi_{j}\left(x_{\alpha}\right)\ . \tag{B.11}$$

In contrast to ordinary least squares, maximum likelihood also gives us an estimate of the noise parameter β as we also maximize the likelihood with respect to β. We find

$$\frac{1}{\beta_{ML}} = \frac{1}{N}\sum_{\alpha=1}^{N}\left(t_\alpha - w_{ML}\cdot\phi\left(x_\alpha\right)\right)^2 . \tag{B.12}$$

The RHS is intuitively an estimation of the variance of the observations t_n about the predictions $y\left(x_n; w_{ML}\right)$. Maximum likelihood therefore ascribes all the residual deviations to noise.

Finally, the *predictive distribution* for t given a value x is

$$p\left(t|x\right) = \mathcal{N}\left(t|w_{ML}\cdot\phi\left(x\right), \beta_{ML}\right) . \tag{B.13}$$

B.2 Maximum Posterior (MAP)

We have so far defined a likelihood function and provided a prescription for extracting predictions for t given values of x. In practice, however, this is equivalent to ordinary least squares regression and comes with all the attendant problems of overfitting. Further, we are still very far away from a Bayesian framework for regression.

Having introduced the likelihood function, we next introduce the *prior distribution* over ws. This models our *a priori* uncertainity in our knowledge of model parameters in terms of a joint probability distribution from which the $M+1$ variables w are drawn. Another way of thinking about the prior distribution is that it models our *a priori* expectations regarding what values our model parameters ws could take. In the following we will assume that the ws are drawn from a Gaussian distribution [22].

$$p\left(w\right) = \mathcal{N}\left(w|m_0, S_0\right) \equiv \mathcal{N}\left(w|0, \alpha^{-1}\mathbb{I}\right) . \tag{B.14}$$

Then, using the Bayes theorem, the posterior probability is proportional to the product of the prior and the evidence.

$$\begin{aligned} p\left(w|t, x, \beta\right) &\propto p\left(t|x, w, \beta\right) \times p\left(w\right) \\ &\sim \exp\left[-\frac{\beta}{2}\sum_{\alpha=1}^{N}\left(t_\alpha - w\cdot\phi\left(x_\alpha\right)\right)^2 - \frac{\alpha}{2}\sum_{j=0}^{M} w_j^2\right] . \end{aligned} \tag{B.15}$$

A different prescription for extracting the values of w is to postulate that this posterior probability distribution is maximized at the observed values of w, i.e. that the observed values of w are the most probable ones given the observed data and our prior beliefs. This prescription is called the *maximum posterior* (MAP). With the assumptions given above, i.e. the form of the prior distribution and the likelihood function, the MAP solution in w is equivalent to the one obtained with least square fitting with an $L2$ regularization $\lambda_2 = \frac{\alpha}{\beta}$.

B.3 A Fully Bayesian Treatment

For a fully Bayesian treatment we should evaluate the probability distribution function $p\left(w|t, x, \beta, \alpha\right)$ in accordance with Bayes Theorem, rather than making point estimates for w. Let us expand out

[22] define conjugate distributions

the exponent and gather the terms linear and quadratic respectively in w. The term quadratic in w is

$$\begin{aligned}\sum_{i,j=1}^{M} w_i \left(-\frac{\alpha}{2}\delta_{ij}\right) w_j - \frac{\beta}{2}\sum_{i,j=1}^{M}\sum_{\alpha=1}^{N} w_i\phi_i\left(x_\alpha\right)\phi_j\left(x_\alpha\right)w_j \\ = -w^T\frac{1}{2}\left(\alpha\mathbb{I}+\beta\Phi^T\Phi\right)w\,,\end{aligned} \tag{B.16}$$

while the term linear in w is

$$\beta\sum_{\alpha=1}^{N}\sum_{i=1}^{M} w_i\phi_i\left(x_\alpha\right)t_n = \beta w^T\Phi^T\cdot\vec{t}. \tag{B.17}$$

Compare with the exponent of $\mathcal{N}\left((w|\mu,\Sigma\right)$, which is

$$-\frac{1}{2}\left(x-\mu\right)^T\sigma^{-1}\left(x-\mu\right) = -\frac{1}{2}x^T\Sigma^{-1}x + x^T\Sigma^{-1}\mu + \text{const}\,. \tag{B.18}$$

Then the posterior is

$$p\left(w|\vec{t}\right) = \mathcal{N}\left(w|m_N,S_N\right)\,, \tag{B.19}$$

where m_N and S_N

$$\begin{aligned} S_N^{-1} &= \alpha\mathbb{I}+\beta\Phi^T\Phi\,,\\ m_N &= \beta\,\Phi^T\cdot\vec{t}.\end{aligned} \tag{B.20}$$

We could have just as well started with the slightly more general prior $\mathcal{N}\left(w|m_0,S_0\right)$ and the computation goes through as before.

Finally, having obtained the posterior, we can compute the predictive function

$$p\left(t|\vec{t},\vec{x},\alpha,\beta\right) = \int p\left(t|w,\beta\right)p\left(w|\vec{t},\vec{x},\alpha,\beta\right)\,. \tag{B.21}$$

Here

$$p\left(t|w,\beta\right) = \mathcal{N}\left(t|y\left(\vec{x},\vec{w}\right),\beta^{-1}\right)\,, \tag{B.22}$$

which was our starting assumption. The integral over w is now Gaussian and we finally obtain

$$p\left(t|x,\vec{x},\vec{t},\alpha,\beta\right) = \mathcal{N}\left(t|m_N^T\cdot\phi\left(x\right),\sigma_N^2\left(x\right)\right)\,, \tag{B.23}$$

where m_N is given in (B.20) and σ_N^2 is

$$\sigma_N^2 = \frac{1}{\beta} + \phi\left(x\right)^T\cdot S_N\cdot\phi\left(x\right)\,. \tag{B.24}$$

The first term is a resultant of the intrinsic noise in data, and the second term arises from the uncertainty in determining the parameters w. We therefore see that the Bayesian treatment of linear regression automatically gives us an estimate of the error made in the prediction for t given an input x.

C Entropy and Cross-Entropy from Information Theory

This section is a quick overview of the information-theoretic definitions of entropy, cross-entropy and their connection with the statistical learning quantities that we have dealt with thus far. Some heuristics are helpful in order to characterize the amount of information contained in the occurence of an event. Firstly, we expect improbable, i.e. surprising, events to be more informative than very probable events. To take this expectation to its extremes, there is zero information in a certain event. Next, if an event occurs twice, then that is twice the information as compared to the event occuring once only. This motivates us to propose that the information $I(x)$ contained in the occurence of an event x with probability $p(x)$ is simply

$$I(x) = -\ln p(x) \; . \tag{C.1}$$

The negative sign makes the information positive as $p(x) \in [0,1]$. Next, if x is a random variable valued in the set X and endowed with the probability distribution $p(x)$, the expectation value of the information is the *Shannon Entropy*, given by

$$H_p = \mathrm{E}\left[-\ln p(x)\right] = -\sum_{x\in X} p(x) \ln p(x) \; . \tag{C.2}$$

If the distribution is approximately deterministic, i.e. $p(x_o) \approx 1$ for some $x_o \in X$ and $p(x) \approx 0$ otherwise then $H_p \approx 0$. By the same token, let there be two probability distributions $p(x)$ and $q(x)$ defined over a random variable $x \in X$. The *Kullback-Leibler* divergence

$$D(p||q) = \mathrm{E}\left[-\ln \frac{q(x)}{p(x)}\right] = -\sum_{x\in X} p(x) \ln \frac{q(x)}{p(x)} \tag{C.3}$$

encodes the extent to which q is different from p; note that $D(p||q)$ is not symmetric in p and q. This quantity would be minimum, and zero, if q were equal to p. See this Wikipedia page for proofs. It is natural to decompose the KL divergence as

$$D(p|q) = -\sum_{x\in X} p(x) \ln \frac{q(x)}{p(x)} \equiv H(p,q) - H(p) \; , \tag{C.4}$$

where we have defined the *cross-entropy*

$$H(p,q) = H(p) + D(p||q) = -\sum_{x\in X} p(x) \ln q(x) \; . \tag{C.5}$$

The minimum of $H(p,q)$ in q is the same as the minimum of $D(p|q)$ in q, *viz.* $q = p$. Let us now apply these ideas to the binary classification problem, where we are given data

$$\mathcal{D}_{train} = \{(x_\alpha, t_\alpha)\} \; . \tag{C.6}$$

We have the *empirical* joint probability distribution

$$p(x,t) = \frac{1}{N} \qquad \forall \qquad (x,t) \in \mathcal{D}_{train} \, . \tag{C.7}$$

We seek to explain these observed data by postulating that they are Bernoulli distributed, i.e. the probability of observing a pair (x,t) is given by

$$q(x,t) = p(C_1|x;w)^t \left(1 - p(C_1|x;w)\right)^{1-t} \; , \tag{C.8}$$

where w are tunable parameters. Then the cross-entropy between the empirical distribution and the hypothetical distribution is

$$H(p,q) = -\sum_{\alpha=1}^{N} \frac{1}{N} \left[t \ln p\left(C_1|x;w\right) + (1-t) \ln \left(1 - \ln p\left(C_1|x;w\right)\right)\right] , \tag{C.9}$$

which we can rewrite as

$$H(w) = -\frac{1}{N} \sum_{\alpha=1}^{N} t \ln p\left(C_1|x;w\right) + (1-t) \ln \left(1 - \ln p\left(C_1|x;w\right)\right) . \tag{C.10}$$

Therefore, minimizing the information-theoretic cross-entropy (C.5) by tuning parameters w of the hypothetical distribution (C.8) is precisely equivalent to minimizing the cross-entropy loss (5.13) or the corresponding log likelihood function. Viewed from this point of view there is no reason to especially single out (5.13) or its multinary generalization as the cross-entropy as opposed to the mean squared loss (3.5). The mean squared error is in fact the cross-entropy between the empirical distribution (C.7) and the postulated distribution (B.7)

$$q(x,t) = \mathcal{N}\left(t|w \cdot \phi(x), \beta^{-1}\right) . \tag{C.11}$$

As before, the principle of maximum likelihood estimation is equivalent to minimizing the cross-entropy between the empirical and the postulated distribution, i.e. requiring that the postulated distribution mimic the empirical one as closely as possible.

References

[1] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2019.

[2] Francois Chollet. *Deep learning with Python.* Simon and Schuster, 2021.

[3] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[4] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[6] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre GR Day, Clint Richardson, Charles K Fisher, and David J Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics reports*, 810:1–124, 2019.

[7] Yang-Hui He. The calabi-yau landscape: From geometry, to physics, to machine-learning. *arXiv preprint arXiv:1812.02893*, 2018.

[8] Fabian Ruehle. Data science applications to string theory. *Physics Reports*, 839:1–117, 2020.

[9] Yang-Hui He. Universes as big data. *International Journal of Modern Physics A*, 36(29):2130017, 2021.

[10] Yang-Hui He. Machine-learning mathematical structures. *International Journal of Data Science in the Mathematical Sciences*, pages 1–25, 2022.

[11] Jiakang Bao, Yang-Hui He, Elli Heyes, and Edward Hirst. Machine learning algebraic geometry for physics. *arXiv preprint arXiv:2204.10334*, 2022.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[13] Daniel Krefl and Rak-Kyeong Seong. Machine learning of calabi-yau volumes. *Physical Review D*, 96(6):066014, 2017.

[14] Fabian Ruehle. Evolving neural networks with genetic algorithms to study the string landscape. *Journal of High Energy Physics*, 2017(8):1–20, 2017.

[15] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[16] Marvin L Minsky and Seymour Papert. Perceptrons and pattern recognition. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1967.

[17] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[18] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[19] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[20] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[21] Leslie N Smith. A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*, 2018.

[22] David H Wolpert. The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390, 1996.

[23] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[24] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.