Contents

Part I

The theory of deep learning

1 Fundamental Ideas

1.1 Introduction

We assume familiarity with traditional machine learning, basic probability and statistics. We look at why a different perspective is needed and why deep learning is a suitable alternative.

1.2 Traditional Machine Learning

The older/traditional way of applying machine learning consisted of the following steps:

1. **Feature Extraction:** Features which can be used to discriminate between classes is identified. This step usually requires in-field knowledge about the problem.

2. **Model Selection:** A model is selected which trains on the extracted features. Ensable methods can be used to boost performance.

3. **Cross-validation/Testing:** The model is tested/cross-validated on withheld data to check accuracy and tune hyperparameters.

The drawbacks of this approach are:

1. **Feature Extraction:** This step requires in-field knowledge. It is very difficult to study a whole new branch of knowledge for a single problem.

2. **Amount of Data:** In the current era, the amount of data sometimes is simply so large that it is hard to extract features manually.

3. **Unorganized Data:** Feature extraction is hard in unorganized data (such as a text corpus or media inputs like images, audio and videos).

1.3 The idea behind a neural network: An intuitive perspective

The idea is to let the machine learn the important features by itself. For example consider the problem of recognizing handwritten digits like in figure 1.1. The machine learns to recognize easy features like say a straight line(highlighted in blue), curved arc(highlighted in red) and circles(highlighted in green) and how those features combine(Like how two circles form an 8).

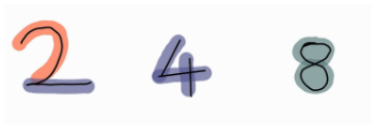


Figure 1.1: Simple features present in handwritten digits

To make an algorithm that can do this we take inspiration from one of the best pattern learning devices in the world: The human brain*. We construct an artificial neuron called a perceptron. Our idea is each perceptron is responsible for recognizing a single feature: It gives a high output whenever a feature is present and a low output when it is absent. So if we have multiple neurons combined, we will be able to recognize complex features that contains many simpler features that the other neurons have identified.

*Taking inspiration from the brain is a repeated theme in deep learning. Those inspirations helped us come up with CNNs and attention mechanisms

1.4 Ideas behind a neural network: A mathematical perspective

From a mathematical point of view, there exists a latent space from where the dataset is sampled from. We model the decision boundary in this space using a parametric equation. Then we use already existing data to tune the parameter so that our modelled decision boundary is an estimate of the actual decision boundary.

1.5 Comparison between traditional ML and deep learning

Traditional ML models show better prediction when the amount of features involved is small. Features can be individually engineered and interpreted. Moreover, such models often provide more transparency on ow each feature is used and should be preferred when the question of how the machine a particular conclusion becomes important. Examples include medical domains or when there is a question of ethics involved.

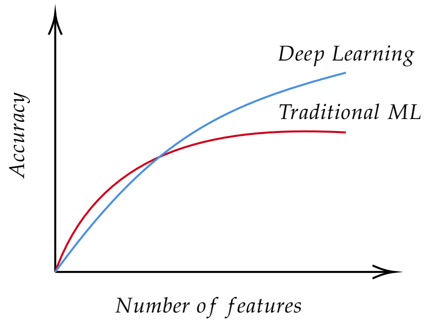


Figure 1.2: Comparison of accuracy between deep learning and traditional ML methods.

Deep learning models are better when data is unstructured or there are a lot of features which need to be considered. With proper construction and training almost any decision boundaries can be learned.

1.6 The perceptron

As mentioned before, a perceptron can be thought to be an artificial neuron. We make a simplification and assume that each perceptron is responsible for identifying some pattern P. A scheme of what a perceptron looks like is given in 1.3

We assume the perceptron returns a high value when it detects P. The inputs to a perceptron can be features from known observation or outputs of other neuron. Let the inputs be $x_1, x_2 \dots x_n$. We arrange them neatly in a vector $X = [x_1, x_2, x_3 \dots x_n]$. Each of those x_is can be thought to be the presence and absence of a simpler feature. We take a weighted sum of those inputs to get $s = \sum_{1 \le i \le n} w_i x_i + w_0$. The intuition is the magnitude of w_i is a measure of the importance of feature x_i and the sign is the direction in which x_i affects the feature which the perceptron is detecting. For example, if the perceptron is detecting if the input is 8 and x_i is the output from another perceptron that detects if a straight line is present then w_i will be negative: there is no straight line in 8. On the other hand, x_j is the output from another perceptron that detects if a circle is present then w_i will be positive: there are two of them in 8. w_0 is just a centering constant. The output of the perceptron will be $y = \sigma(s)$, where σ is known as the activation function.

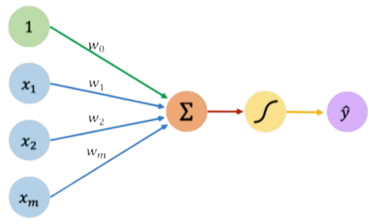


Figure 1.3: Schematic diagram of a perceptron,*Src: MIT Introduction to Deep Learning,6.S191,Lec-1*

1.7 *Chaining Neurons: A simple neural network*

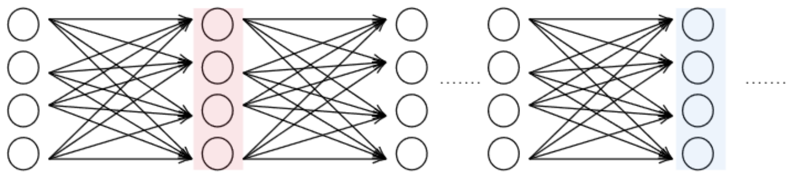


Figure 1.4: A simple neural network

A very simple neural network is show in 1.4. Each circle represents the output of a perceptron while the arrow pointing to it are the inputs. Each row of neuron is(for example, the blue or the red box) called a layer and the layers between the input and output layer are called hidden layers. The layers on the left(For example, the red layer) learns the simpler features while the layers on the right(For example, the blue layer) learns more complex features, which is composed of multiple simpler features.

1.8 *Non-linearity of the activation function*

Ideally, we want our activation function to be non-linear. A simple calculation shows if the activation function for all perceptrons are linear then even if you use multiple layers the output will be a linear combination of the input. Therefore, we use non-linear activation functions to capture non-linear dependencies between the input features/ simpler patterns. From a more mathematical perspective, we wish to transform the latent space in a way so that the decision boundaries are linear. This is a common idea when we deal with the question of on which "side" of a hypersurface does a point lie in. For example, if our latent surface is $\mathbb{R}^2$ and the decision boundary is given by some nicely parameterized curve γ (Some examples of a nice γ are $x^2 + y^2 - r^2 = 0, mx + c - y = 0, p(x) - y = 0$ where p is polynomial) then the two sides of gamma are given by $\gamma^{-1}(-\infty, 0)$ and $\gamma^{-1}(0, \infty)$. An example showcasing this is given in 1.5

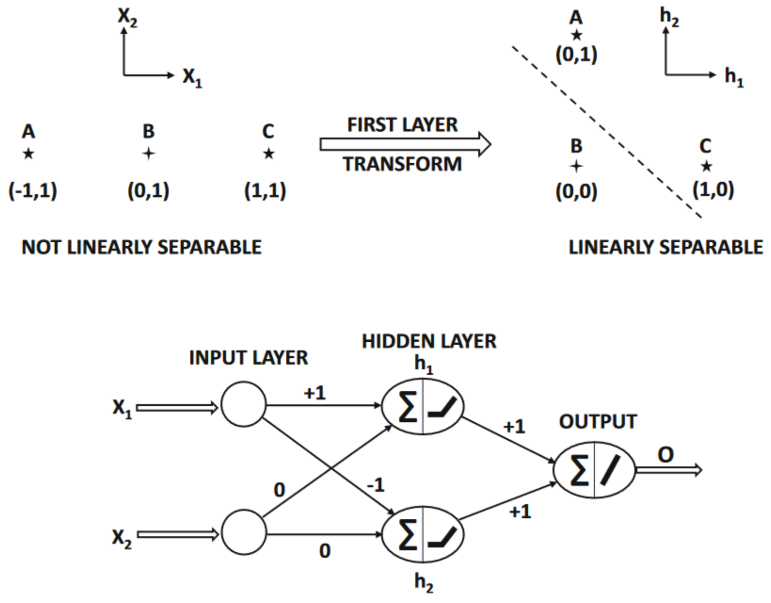


Figure 1.5: A representation of how a non-linear activation function transforms the latent space to give linear boundaries(Aggarwal et al., 2018)

1.9 Inputs and outputs to a neural network

1.9.1 Classification problem with tabular dataset

Suppose you have an observation where given features $X = [x_1, x_2 \ldots]$ you need to predict the label Y. First we replace all categorical x_i and Y with their one hot representation*. If X can take one l labels (number of possible choices of Y), then the second to last layer and the last layer both have l nodes. If the output from the second to last layer is

*Given categories $c_1, c_2, c_3 \ldots c_m$, if x_i belongs to category c_k then in a one hot representation, we represent x_i by e_k where e_k is the k^{th} vector in the standard basis

y, at the last layer we take a softmax defined by:

$$\hat{Y}_i = \frac{\exp(y_i)}{\sum_{1 \leq i \leq l} \exp(y_i)}$$

The softmax function maps $\mathbb{R}^n \rightarrow [0,1]^n$ and the interpretation is $\hat{Y}_i = P(X \text{ has the label } i)$[Note, by definition $\hat{Y}_i \in [0,1]$ and $\sum \hat{Y}_i = 1$]. Softmax is quite expensive to compute. Depending on the problem, we might replace it with other scoring system where a higher $\hat{Y}_i$ in the output layer means a higher probability that X is in class i.

1.9.2 *Problems where the input is unstructured data*

If the data is unstructured, we use embedding mechanisms to represent them as points in some $\mathbb{R}^n$ in such a way that similar entries are closer together. Then we proceed as usual.

1.9.3 *Regression tasks*

In regression problems, the output $\hat{Y}$ is simply an estimate of Y.

2 How to train a neural network

2.1 Introduction

Once we have made our model of a neural network, we would like to train it. The process of training involves tuning the set of weights $W = [w_0, w_1, w_2 \ldots]$ associated with each perceptron. To do so we shall give the neural network a rigorous mathematical structure and look at methods to efficiently adjust our weights.

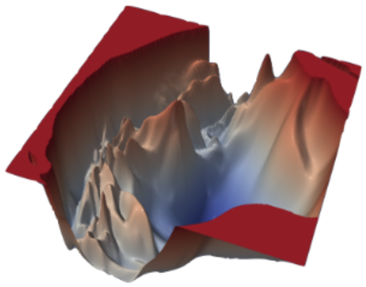


Figure 2.1: A slice of the loss landscape(A graph of $\mathcal{L}$ vs (w_i, w_j)) in ResNet(an example of a type of neural network). Note that it is quite hard to find the minima in this.(Li et al., 2018)

2.2 The loss functions

The loss function can be thought to be a measure of the efficiency of a neural network. Suppose we have a set of observations $X_0 = [X_1, X_2, \ldots X_n]$ with known labels/values $Y_0 = [Y_1, Y_2 \ldots Y_n]$. Assume our neural network gives predictions $\hat{Y}_0 = [\hat{Y}_1, \hat{Y}_2 \ldots Y_n]$. Then loss function $\mathcal{L}(\hat{Y}_i, Y_i)$ calculates how off our prediction was from the actual label. We define the total loss as $\mathcal{L}(\hat{Y}_0, Y_0) = \sum \mathcal{L}(\hat{Y}_i, Y_i)$. Generally, we use mean square error(MSE) as the loss function for regression problems and categorical cross entropy for classification problems. That being said, in more complex network, the loss function may be more complicated. For example, a common problem in computer vision is to identify objects in an image. It is found that it is easier to answer the question in two parts:

1. Is there an object present in the image?

2. If yes, where is it in the image?

As we can understand, the first question is a classification problem whereas the second question is a regression problem(assuming we give our answer as coordinates). Now that we have a measure of how good our neural network is, we can think of training to be tuning the parameters to minimise loss.

2.3 Gradient Descent

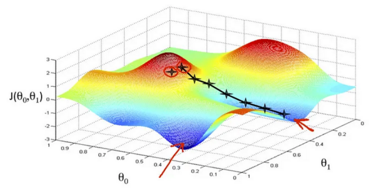


Figure 2.2: Gradient descent, Src: https://towardsdatascience.com/an-intuitive-explanation- of-gradient-descent-83adf68c9c33

To find the correct set of weights, we use a greedy approach. We check the surrounding landscape of the weight(i.e. calculate the gradient) and take a step in the direction which leads to maximum decrease in L. This is an iterative process. Mathematically, for the weight w_i we have the following update rule:

$$w_i \to w_i - \eta \frac{\partial \mathcal{L}}{\partial w_i}$$

η is known as the learning rate. Fixing η is quite tricky: too large and it shoots part the minima, too small and it never converges. The best way to do it is to use an adaptive learn rate. Some methods(parametric, non-parametric and hybrid are discussed later, once we cover back propagation)

2.4 Neural network as a directed acyclic graph (DAG)

We look at neural network as a DGA. Each variable (output of perceptron, weight and feature of input) is a vertex. An edge connects vertex v_i to v_j if v_i is directly needed for the computation or updation of v_j. For a perceptron with output $x = \sigma(w_0 + \sum w_i x_i)$, there are edges from all x_i and w_i to x. In some cases, η is not a constant. In that case, there are edges from the parameters η depend on to x.

2.5 Reverse mode auto differentiation

The general algorithm that is used for gradient descent is called backpropagation. In it's most basic implementation the running time is exponential in the number of layers, which is undesirable. So we talk about a slightly different implementation known as reverse mode auto differentiation*. Each iteration takes place in two steps: the forward phase and the backward phase.

*As it turns out, people in control theory were using this way before this was independently invented for use in deep learning. Kinda shows how low inter topic information sharing is

2.5.1 Forward phase

In the forward phase, the algorithm simply calculates the total loss $\mathcal{L}(\hat{Y}_0, Y_0)$. This is called the forward phase as we calculate along the direction of the edge.

2.5.2 Backward phase

In that backward phase we calculate the gradients and update weights. We calculate the gradients by repeatedly applying chain rule. If there is an edge from v_i to v_j then $\frac{\partial v_j}{\partial v_i}$ can be calculated directly. If they don't have an edge connecting them, we take the product of the derivatives along the edges on a path and then take the sum along all the paths. A small example is given in 2.3

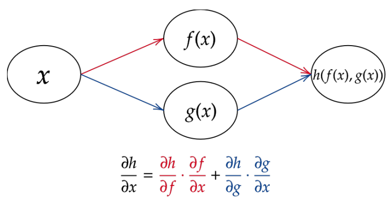

x
$f(x)$
$g(x)$
$h(f(x), g(x))$
$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial x} + \frac{\partial h}{\partial g} \cdot \frac{\partial g}{\partial x}$

Figure 2.3: A small example of chain rule application

At this point it should be obvious that this process is going to take exponentially long the deeper the network is: there are simply too many paths. Here we use dynamic programming. Consider the example in 2.8

We see that certain terms are repeated in the expression. This is because parts of the path(shown in colored arrows) is repeated. Therefore, if we can store the gradients for some edges then we don't need to calculate all the terms every time. This significantly reduces computation and makes the algorithm practical to implement. This is called the backward phase as the gradients are calculated against the direction of the edges.

2.6 *Gradient Descent Strategies*

2.6.1 *Stochastic Gradient Descent*

Instead of calculating the total loss, we calculate the loss from a randomly picked sample(or a batch in case of batch gradient descent). Since we are no longer calculating the total loss over all data points, this decreases the computational time. A proper analogy might be instead of taking slower but confident steps, we take faster but less-confident steps.

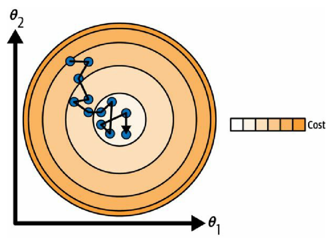


Figure 2.4: Stochastic Gradient Descent. Note that we don't take steps on the best direction, but it is faster.(Géron, 2022)

2.6.2 *Normalization*

Normalizing features is a way to make the descent smoother. It essentially lowers gradient in directions orthogonal to the minima. This also eases setting the learning rate. If one feature varies between 0 to 255 and other between 0 and 1, it can be very difficult to set a base learning rate. Normalizing features solves this problem.

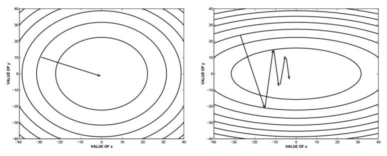


Figure 2.5: Normalization(Aggarwal et al., 2018)

2.6.3 *Momentum*

A Momentum term might be used in gradient descent where consideration is made for a moving average "velocity" of the descent. Such strategies are particularly helps when there are local minimas and flat regions. Also helps when there is a lot of "zig-zag" but the descent on an average heads in a certain direction. We can improve on this by doing some scout ahead. This is known as Nestov momentum, and it helps as knowing what's coming up ahead further helps in correcting the direction of descent.

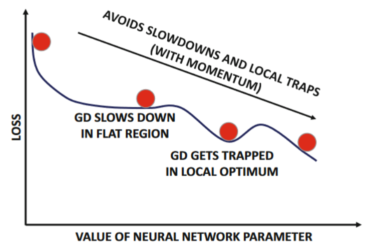


Figure 2.6: Momentum preventing getting stuck at local minima(Aggarwal et al., 2018)

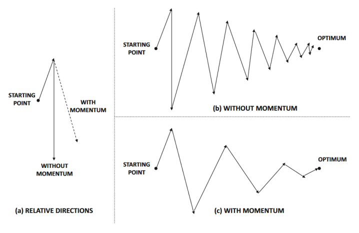
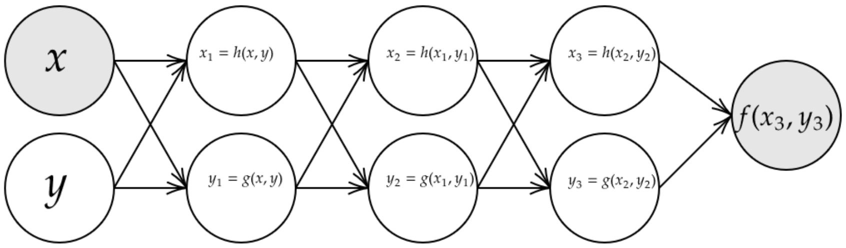


Figure 2.7: Momentum reducing zig-zag(Aggarwal et al., 2018)



$$
\begin{aligned}
\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial x_3} \cdot \frac{\partial x_3}{\partial x} + \frac{\partial f}{\partial y_3} \cdot \frac{\partial y_3}{\partial x} \\
&= \frac{\partial f}{\partial x_3} \cdot \left(\frac{\partial x_3}{\partial x_2} \frac{\partial x_2}{\partial x} + \frac{\partial x_3}{\partial y_2} \frac{\partial y_2}{\partial x} \right) + \frac{\partial f}{\partial y_3} \cdot \left(\frac{\partial y_3}{\partial x_2} \frac{\partial x_2}{\partial x} + \frac{\partial y_3}{\partial y_2} \frac{\partial y_2}{\partial x} \right)
\end{aligned}
$$

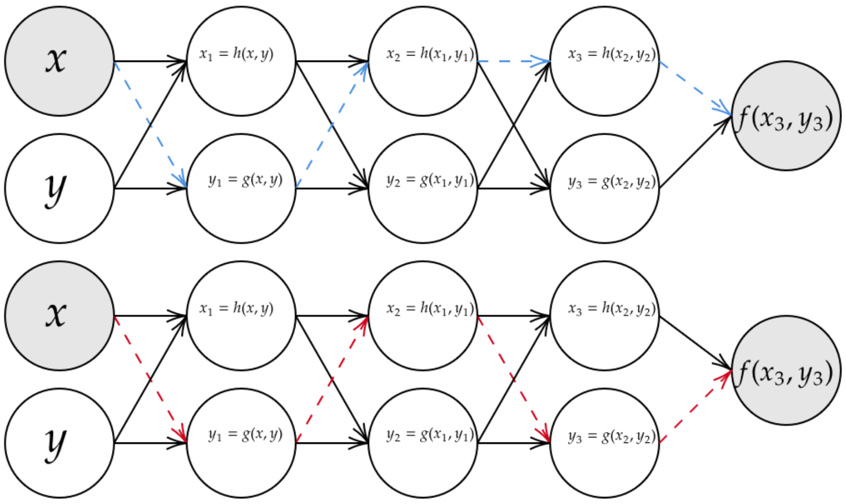


Figure 2.8: Chain rule in deep networks

2.7 Setting the learning rate

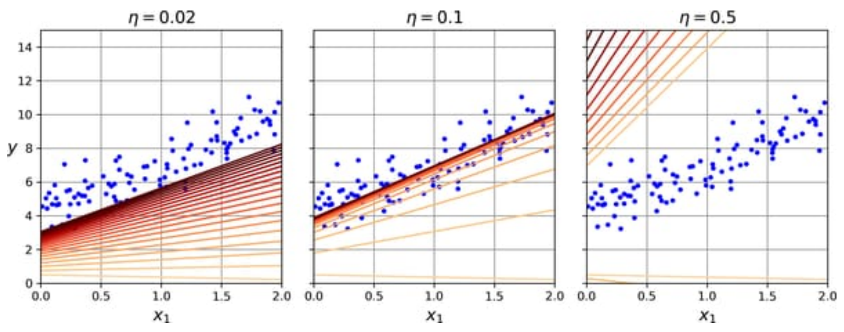


Figure 2.9: Why is setting a proper learning rate important(Géron, 2022)

2.7.1 Adaptive learning rate

One way to take care of this problem is to let different parameters have different learning rates. The idea is that parameters with large partial derivatives are often oscillating and zigzagging, whereas parameters with small partial derivatives tend to be more consistent but move in the same direction. Algorithms include AdaGrad, RMSProp and AdaDelta. Parametric algorithms combined with momentum considerations also exist: RMSProp with Nesterov Momentum, ADAM and it's variants like AdaMax, Nadam and AdamW. A quick comparison table is available at (Géron, 2022)

2.7.2 Learning rate scheduling

We have the learning rate high at first so that it gets a relative idea about where the minima lies, and then we lower it to find it with more accuracy. Scheduling is done on numbers of iterations completed(each iteration is also called an epoch)

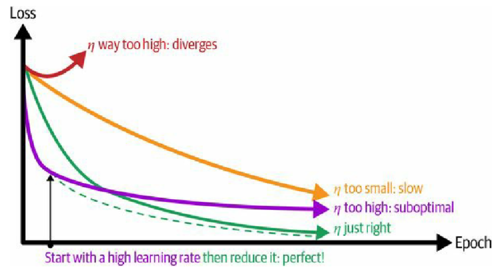


Figure 2.10: Learning rate scheduling(Géron, 2022)

2.7.3 One cycle scheduling

The basic idea behind one cycle scheduling is to start with a low learning rate, gradually increase it to a maximum value, and then decrease it again to a low value. This approach helps the network explore a wide range of learning rates, allowing it to quickly converge to a good solution and potentially escape from local minima. By using a cyclical learning rate schedule, one cycle scheduling aims to strike a balance between exploration and exploitation in the learning process. It enables the network to quickly explore a wide range of learning rates at the beginning and then gradually refine its weights as the learning rate decreases.

2.8 *Choosing Activation function*

The loss function and the gradients propagated backwards depend on the activation functions used. A bad activation function(for example, a function which is not differentiable everywhere) can make training very hard. We first look at some

Bibliography

Charu C Aggarwal et al. Neural networks and deep learning. *Springer*, 10(978):3, 2018.

Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. *Advances in neural information processing systems*, 31, 2018.