Contents

I The theory of deep learning **3**

1 Fundamental Ideas **5**
- 1.1 Introduction . . . 5
- 1.2 Traditional Machine Learning . . . 5
- 1.3 The idea behind a neural network: An intuitive perspective . . . 6
- 1.4 Ideas behind a neural network: A mathematical perspective . . . 6
- 1.5 Comparison between traditional ML and deep learning . 6
- 1.6 The perceptron . . . 6
- 1.7 Chaining Neurons: A simple neural network . . . 7
- 1.8 Non-linearity of the activation function . . . 7
- 1.9 Inputs and outputs to a neural network . . . 8
  - 1.9.1 Classification problem with tabular dataset . . . 8
  - 1.9.2 Problems where the input is unstructured data . 9
  - 1.9.3 Regression tasks . . . 9

2 How to train a neural network **11**
- 2.1 Introduction . . . 11
- 2.2 The loss functions . . . 11
- 2.3 Gradient Descent . . . 12
- 2.4 Neural network as a directed acyclic graph (DAG) . . . 12
- 2.5 Reverse mode auto differentiation . . . 12
  - 2.5.1 Forward phase . . . 12
  - 2.5.2 Backward phase . . . 12
- 2.6 Gradient Descent Strategies . . . 13
  - 2.6.1 Stochastic Gradient Descent . . . 13
  - 2.6.2 Normalization . . . 13
  - 2.6.3 Momentum . . . 13
  - 2.6.4 Gradient Clipping . . . 15
- 2.7 Setting the learning rate . . . 15
  - 2.7.1 Adaptive learning rate . . . 15

2.7.2 Learning rate scheduling . . . 15
2.7.3 One cycle scheduling . . . 15
2.8 Choosing Activation function . . . 16
2.8.1 Dead Neurons . . . 16
2.8.2 Presence of local minimas . . . 16
2.8.3 Exploding and vanishing gradients . . . 16
2.8.4 Classical Activation Functions . . . 17
2.8.5 Newer Activation functions . . . 17
2.9 Weight Initialization . . . 17
2.10 Aspects of a deep neural network . . . 17
2.10.1 Skip connections . . . 18
2.10.2 Maxout Networks . . . 18
2.11 Setting hyperparameters . . . 18

II Deep Learning in NLP **21**

3 Why is language a pain to model? And how do we model them. 23
3.1 Language and ideas . . . 23
3.2 Why is NLP hard? . . . 23
3.3 Context and Lexical semantics . . . 24
3.4 Language as a Markov process . . . 25
3.5 Language as a stochastic process . . . 25
3.6 Crude early models and their shortcomings . . . 25

4 Word Embeddings **27**
4.1 Why vectors? . . . 27
4.2 Data Preparation . . . 27
4.3 One-hot Representation of Words and Context . . . 27
4.4 Continuous bag of words model(CBOW) . . . 28
4.4.1 Interpretation on a two word window . . . 30
4.5 Skip-Gram . . . 30
4.6 Negative sampling . . . 31
4.7 Contrastive Estimation . . . 32
4.8 SVD to learn word representations . . . 32
4.9 GloVe Representations . . . 33
4.10 Other embeddings . . . 33

III Sequence Modelling **35**

Bibliography **39**

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).

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.

Figure 1.1: Simple features present in handwritten digits

*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.

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.

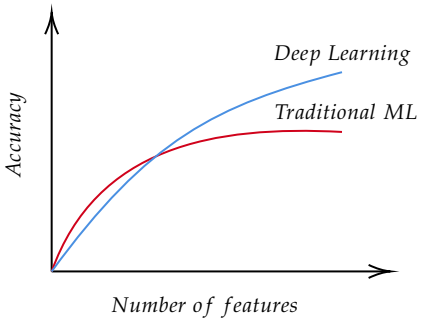


Figure 1.2: Comparison of accuracy between deep learning and traditional ML methods.

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 $s = \sum_{1 \leq i \leq n} w_i x_i + w_0$. 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.

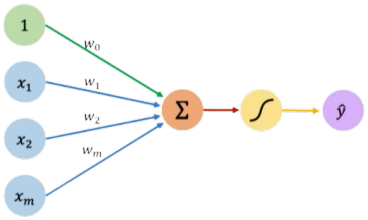


Figure 1.3: Schematic diagram of a perceptron,*Src: MIT Introduction to Deep Learning,6.S191,Lec-1*

1.7 *Chaining Neurons: A simple neural network*

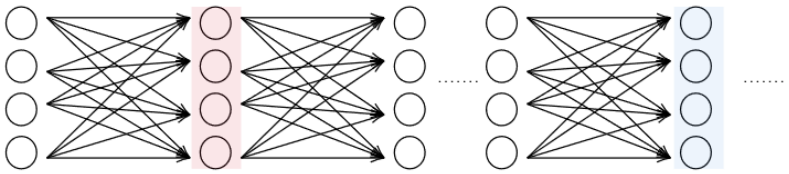


Figure 1.4: A simple neural network

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

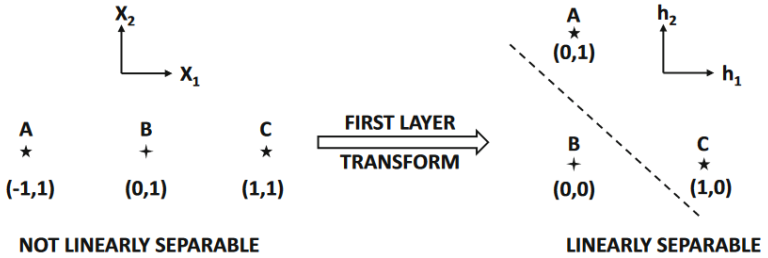
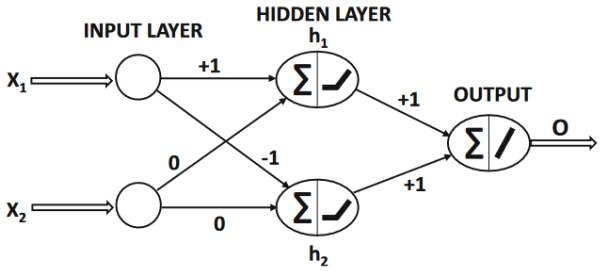


Figure 1.5: A representation of how a non-linear activation function transforms the latent space to give linear boundaries(Aggarwal et al., 2018)



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 $X = [x_1, x_2 \dots]$ 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 $c_1, c_2, c_3 \dots c_m$, 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 $\mathbb{R}^n \to [0,1]^n$ 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.

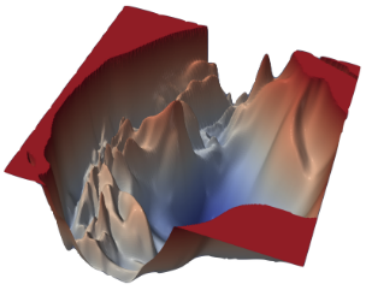


Figure 2.1: A slice of the loss landscape(A graph of $\mathcal{L}$ vs (w_i, w_j)) in ResNet(an example of a type of neural network). Note that it is quite hard to find the minima in this.(Li et al., 2018)

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

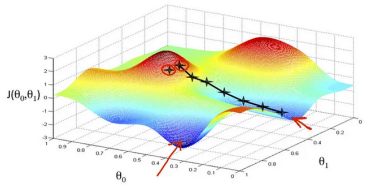


Figure 2.2: Gradient descent, Src: https://towardsdatascience.com/an-intuitive-explanation- of-gradient-descent-83adf68c9c33

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

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.

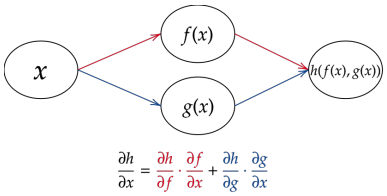


Figure 2.3: A small example of chain rule application

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.

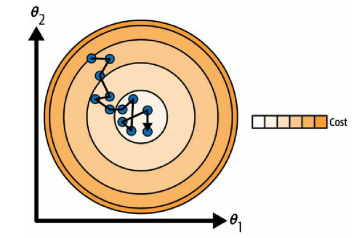


Figure 2.4: Stochastic Gradient Descent. Note that we don't take steps on the best direction, but it is faster.(Géron, 2022)

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.

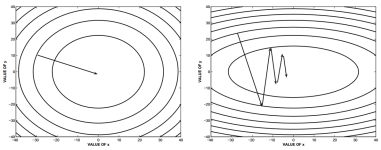


Figure 2.5: Normalization(Aggarwal et al., 2018)

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.

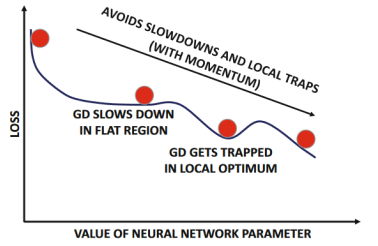


Figure 2.6: Momentum preventing getting stuck at local minima(Aggarwal et al., 2018)

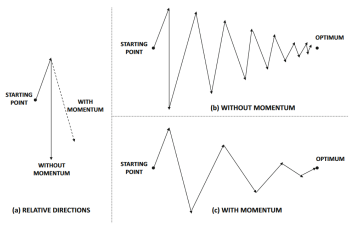
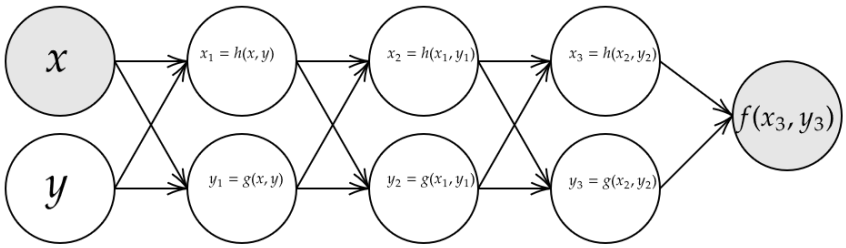


Figure 2.7: Momentum reducing zig-zag(Aggarwal et al., 2018)



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

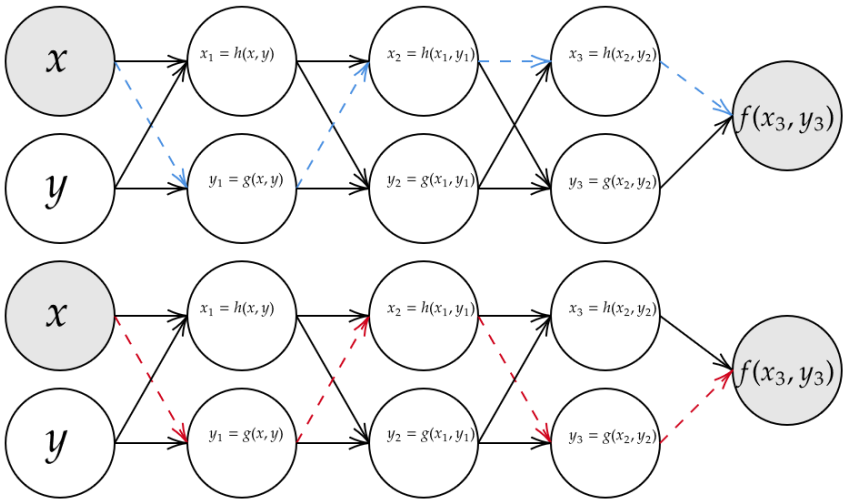


Figure 2.8: Chain rule in deep networks

2.6.4 *Gradient Clipping*

In case one of the components of the gradient is much lager than other we might want to normalize it to control our descent. One way it to just limit it at some maximum value. Other methods include normalizing the whole vector with respect to a norm.

2.7 *Setting the learning rate*

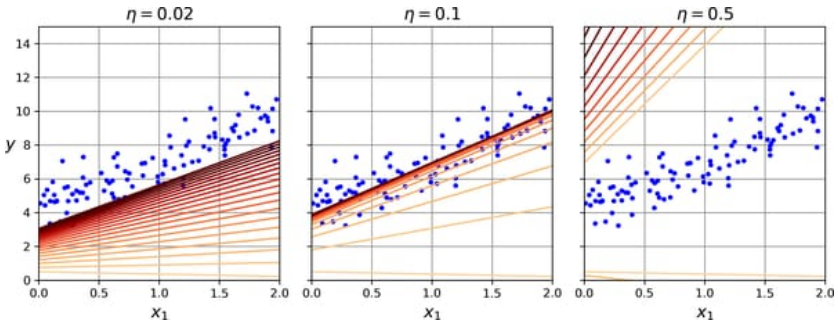


Figure 2.9: Why is setting a proper learning rate important(Géron, 2022)

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 problem that might arise due to poor choice of activation problem(Although, some of those problems might arise even with the correct choice of activation function).

2.8.1 *Dead Neurons*

Many activation functions saturate/become constant in certain regions. This leads to 0 gradient flowing backwards from neurons and neurons connected to those neurons don't get properly trained. Moreover, if weights associated with those neurons are always too high/too low, they always return a high/low value regardless of input. Such neurons are called dead: they might as well not exist, and the network will run without much change.

Those problems suggest that we might want out function to not saturate easily.

2.8.2 *Presence of local minimas*

We don't want local minimas in our loss function: gradient descent algorithms might get stuck over there. We therefore attempt to make sure that our activation function is strictly increasing. On a brighter note, in most cases, a local minima is good enough.

2.8.3 *Exploding and vanishing gradients*

The effect of derivatives magnifies down the layers. If the order of magnitude is above 1(say 10) then it explodes(after 10 layers it will become 10^{10}). If the order of magnitude is below 1(say 1/10) then it vanishes(after 10 layers it will become 10^{-10}). From a more practical viewpoint, this might lead to value overflow if the gradient explodes.

If it is vanishing, then there might not be enough precision to handle the values.

2.8.4 *Classical Activation Functions*

Classically, we used sigmoid, tanh and relu as activation functions of choice. All three of them saturate and can lead to dead neurons. Still, sigmoid is used even nor in LSTMS as it gives an output between 0 and 1.

2.8.5 *Newer Activation functions*

Newer activation functions were created which were modified versions of relu like leaky relu, elu and selu. More exotic activation functions also exist like gelu, swish and mish. In some cases those functions are more expensive to compute, but the neurons are trained fast enough to compensate for the extra computation time and the there is an overall decrease in wall time.

2.9 *Weight Initialization*

We wish to initialize weights randomly. As best explained by Goodfellow(Goodfellow et al., 2016): "Perhaps the only property known with complete certainty is that the initial parameters need to "break symmetry" between different units. If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters. If they have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way".

Initially weights were initialized by picking from a standard normal distribution. But it can be shown that this cause problem when there difference in number of inputs/outputs among nodes. An intuitive analogy is when the amount of change fed to it is different from the change it outputs during gradient descent. In He initialization We initialize by picking values from a normal distribution with a variance of $\sqrt{2/n}$ where n is the number of incoming nodes. We can also use n =average of number of input and output nodes.

2.10 *Aspects of a deep neural network*

While a single hidden layer is enough, making a deep neural network allows us to have more complex decision boundaries with relatively less number of nodes: adding depth gives ore bang for the buck than

adding more nodes in a single layer. But depth comes with its own share of problems. Deeper networks have a harder time converging. Moreover, if activation functions are not combined properly then there might be presence of local minimas. We discuss two methods which can help with this.

2.10.1 *Skip connections*

We might allow nodes from deeper layers to have direct access to nodes from shallower layers while skipping the intermediate layers. This provides a kind of highway for the deeper layers to access simpler features makes sure that only required higher level features are learned. Also, this reduces a bit of the problems from gradient flows as shallower layers doesn't depend only on the shallower layers for gradient updates. They can also get trained before all the middle layers get their weights right.

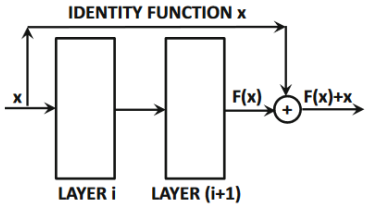


Figure 2.11: Skip connections (Aggarwal et al., 2018)

2.10.2 *Maxout Networks*

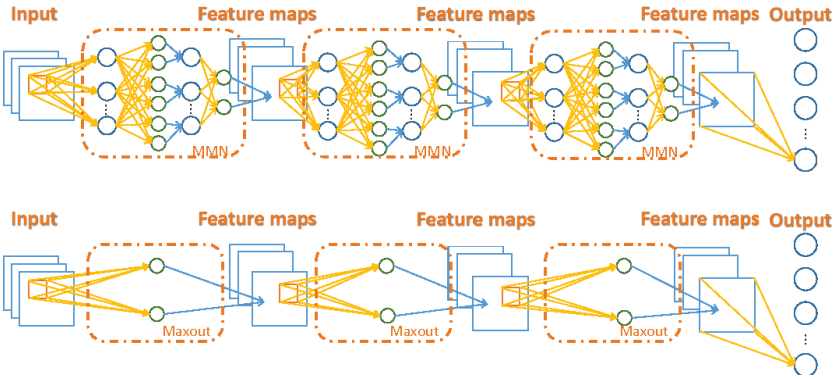


Figure 2.12: Maxout Networks(Sun et al., 2018)

In a certain sense this is type of ensemble method. Two sets of weights W_1, W_2 are trained for each perceptron. The activation is set to $\sigma(\max(W_1 \cdot X_{in}, W_2 \cdot X_{in}) + b_0)$

2.11 *Setting hyperparameters*

Apart from the usual parameters for activation functions, there are other hyperparameters to be set. Those include stuff like number of layers, number of nodes in a layer, presence and nature of skip connections, presence and nature of max-out layers, choice of activation functions, choice of learning rates and schedules. It is not practical to test on the training dataset to prevent overfitting(discussed later) nor is it desired to test on the testing dataset. In such a scenario using the

cross validation set is our best bet. For setting the hyperparameters we simply choose a possible set of values (a point in $\mathbb{R}^n$) for each parameter and try everything on the Grid. We start with coarse grids and then move on to finer ones. The idea is trying all points on the grid by running small number of epochs, see what fits best and then train on those parameters.

Part II

Deep Learning in NLP

3 Why is language a pain to model? And how do we model them.

3.1 Language and ideas

Language arose about $100,000 - 1,000,000$ years ago and that took us from Bronze Age to modern science. From a linguistic point of view, a word is the representation of an idea. Words representing old obsolete ideas are often forgotten, and new words representing new ideas.

3.2 Why is NLP hard?

Written as a person fluent in English, Bengali and Hindi

The short answer is when we speak, we leave out massive chunks of information. A lot of the information conveyed while writing assumes that the reader has prior context and experience to extract meaningful information from the text. It is even more difficult when looking at transcriptions of spoken text: While speaking we use our facial expressions and vocal tones as a part of communication. Some particular difficulties are described below:

1. Homonyms: When we say "He/She is cool" we often mean that a person is calm and collected. We use our prior experience to infer "cool" is not referring to feeling coldness. Ideally, we would like our model to understand such subtleties. More examples include:

   - **Iraqi Head Seeks Arms**: Here head is to be interpreted as leader and not as body part.
   - **Kids Make Nutritious Snacks**: Here make is to be understood as the process of creation and is not meant as a part of.
   - **Miners Refuse to Work after Death**: Here death refers to death of a person and not the state of being dead.

2. For complex sentences we use prior experiences for deciphering meaning. This process is made more complex by the fact that

in English language, a single word can act as different parts of speech. For example,

- **Enraged Cow Injures Farmer with Ax**: the with ax part is describing the farmer and not the cow.
- **Hospitals are Sued by 7 Foot Doctors**: The phrase"7 foot" are two different ideas: 7 describes the number of doctors and foot describes their specialty of work.
- **Students cook and serve grandparents**: Grandparents are served food, they are not the food

More examples are available here:
`https://www.plainlanguage.gov/resources/humor/funny-headlines/`

3. Figures of speech: We use the example of sarcasm. When we use sarcasm we mean the opposite of what we say. We again use prior experience to identify that the writer doesn't mean what they say. For example, consider a very simple case: "Great! Now my tires are flat". We use our prior knowledge that a flat tire (in most cases) is not a desirable thing and so the word "great" is not used in a positive sense. Things get tricker when this is used in more subtle sense.

4. Not all ideas are available in all languages: As a very artificial example, no 16^{th} century Indian language would have a word for potato simply because no one knew what a potato was! Some more examples are available here:
`https://www.babbel.com/en/magazine/untranslatable-01`

5. As someone once said, language is a living thing. New words are always being made to represent new ideas and the internet has accelerated this process. Words like Poggers, KEKW, OC etc. were invented in the era of memes and twitch chat. Usage of unalive (which was not an actual word some time ago) is gaining popularly as the actual word "dead" is often flagged by online platforms.

3.3 *Context and Lexical semantics*

Each word represents an idea, and when we communicate we chain those ideas. This gives rise to the idea of context. We claim a word gets it's meaning from the words surrounding it. For example, if we have a new word **plompyskompy** and we say it is used as follows: "There is an apple **plompyskompy** on the table" you might (again, due to your experience) understand it represents same kind of idea the word "on" represents. You understand this due to the words

surrounding it. This leads us to the idea of Lexical semantics(Ref: https://en.wikipedia.org/wiki/Lexical_semantics)

3.4 *Language as a Markov process*

This will now be a quick discussion. A word gets it's meaning from its surrounding words. The rest of the corpus doesn't matter. What came 100 words ago doesn't matter(unless we are considering what came 100 words ago to be a part of the context of the word). We look at the last few words and by our theory, it should be enough to predict what our next word is going to be. Therefore, if we are looking at a context window of n words then it should be modelled as an n^{th} order Markov chain. Ref:
https://www.cs.princeton.edu/courses/archive/spr05/cos126/assignments/markov.html

3.5 *Language as a stochastic process*

Languages can also be modelled as stochastic process, i.e. we drop the assumption that local context is all that is needed. And it should be noted that the memoryless property of Markov processes are counterintuitive to our claim that experience plays an important role in decoding ideas from sentences. Ref: https://openreview.net/forum?id=pMQwKL1yctf

3.6 *Crude early models and their shortcomings*

Traditionally, a computer understands a word by using a dictionary and looking up synonyms(word with similar meaning), hypernyms(expression of belonging to a more general category), hyponyms(expression of belonging to more specific subclass). A minimal example implemented in NLTK is shown below. Some arrays are truncated for brevity.

```
>>> from nltk.corpus import wordnet as wn
>>> wn.synonyms('car')
[['auto', 'automobile', 'machine', 'motorcar'], ['railcar',
    'railroad_car', 'railway_car'], ['gondola'],
    ['elevator_car'], ['cable_car']]
>>> wn.synsets('car')[:2]
[Synset('car.n.01'), Synset('car.n.02')]
>>> wn.synset('car.n.01').definition()
'a motor vehicle with four wheels; usually propelled by an
    internal combustion engine'
>>> wn.synset('car.n.01').examples()
['he needs a car to get to work']
>>> wn.synset('car.n.01').hypernyms()
[Synset('motor_vehicle.n.01')]
```

```
>>> wn.synset('dog.n.01').hypernyms()
[Synset('canine.n.02'), Synset('domestic_animal.n.01')]
>>> wn.synset('car.n.01').hyponyms()[:5]
[Synset('ambulance.n.01'), Synset('beach_wagon.n.01'),
    Synset('bus.n.04'), Synset('cab.n.03'),
    Synset('compact.n.03')]
```

But this is not a very robust solution: dependency on context and nuances are not resolved(Ex: "crimson" is a synonym of "red" but "the color of an apple is crimson" is a weird sounding sentence). Moreover, language is ever-changing and keeping track of an ever changing

In traditional bag of words approach, words are given a one hot vector representation. For example:

$$\text{"hotel"} = [0, 0, 0, 0, 0, 1, 0, 0, 0]$$
$$\text{"motel"} = [0, 0, 0, 0, 0, 0, 0, 1, 0]$$

But this approach has its own limitations:

1. Vector size increase with increase in vocabulary
2. We understand that the words above are similar, but that similarity is not reflected in such one-hot encoding.

4 Word Embeddings

4.1 Why vectors?

We would like to have a structure

1. that can be modelled using a small(<- this depends on what your idea of small is, but we would like to have a memory efficient approach) number of parameters

2. that has an inbuilt idea of similarity

3. that can be easily manipulated

As it turns out finite dimensional vector spaces are perfect for this task: the number of parameters is the number of dimensions, a metric can be used as notion of similarity, and we already have tools to manipulate them.

4.2 Data Preparation

We need to prepare the data or the next two models. We select a window size n. From the corpus, n consecutive words are sampled. The middle word is the target word and the rest is the context. Each such target word, context pair is a data point.

4.3 One-hot Representation of Words and Context

Once the data preparation is done we will have a set of data points of the form (S, W) where S is the context and W is the word. If the word is in the i^{th} place in the vocabulary V then we represent W the one-hot vector e_i (i^{th} element of the standard basis). If the context contains the words $W_1 = V[\alpha_1], W_2 = V[\alpha_2] \dots W_n = V[\alpha_n]$ then we will represent the context S as $\sum_{i=1}^{n} e_{\alpha_i}$, i.e. the sum of the one-hot encoding of the individual words. For the rest of this discussion I will use the notation $w_i, V[i]$ and e_i interchangeably for words.

4.4 *Continuous bag of words model(CBOW)*

Assume we have a data point(we have taken size 5 for the example, but this should be adjusted as needed) as follows:

Word1 Word 2 Word Word3 Word 4

We model a classification problem where given context {Word1, Word2, Word3, Word4} we wish to classify which word this set provides context for.[Note, when we represent it as a set, we automatically let go the notion of a sequence- hence the name 'bag of words']. For this we shall use a feed forward network described below:

1. **Input :**For context $S \subset V$, we make an input vector v where:

$$v_S = \sum_{w_i \in S} e_i$$

2. **Hidden Layer 1**: This is obtained by multiplying v by a matrix W. We denote:

$$h_S = W v_S$$

   For a single word w_i, $h_i = We_i = i^{th}$ column of W will represent the hidden representation of w_i. Therefore, if our context contains is $S = \{w_{\alpha_1}, w_{\alpha_2}, w_{\alpha_3}, w_{\alpha_4} \dots w_{\alpha_n}\}$ then our hidden representation of the context will be $h_S = W\left(\sum_{k=1}^{n} e_{\alpha_k}\right) = \sum_{k=1}^{n} h_{w_{\alpha_k}}$ (Note how our choice of using a linear space as structure of choice and setting the context as the linear combination of its constituents comes handy here)

3. **Hidden Layer 2**: We use another matrix W' and get

$$y = W' h_S$$

4. **Hidden Layer 3**: This is just a soft max to normalize things:

$$\hat{y} = \text{Softmax}(y)$$

We use cross-entropy as our loss function of choice to train the model. $\hat{y}$ is compared to the one-hot representation of the word.

We have:

$$\hat{y} = Softmax(W'(h_S))$$

But as h_S is independent of all h_w where $w \notin S$. Therefore, we don't compute Wv_s nor do we keep track of all the weights. We look only at those columns of W that correspond to words in the context and don't care about the rest.

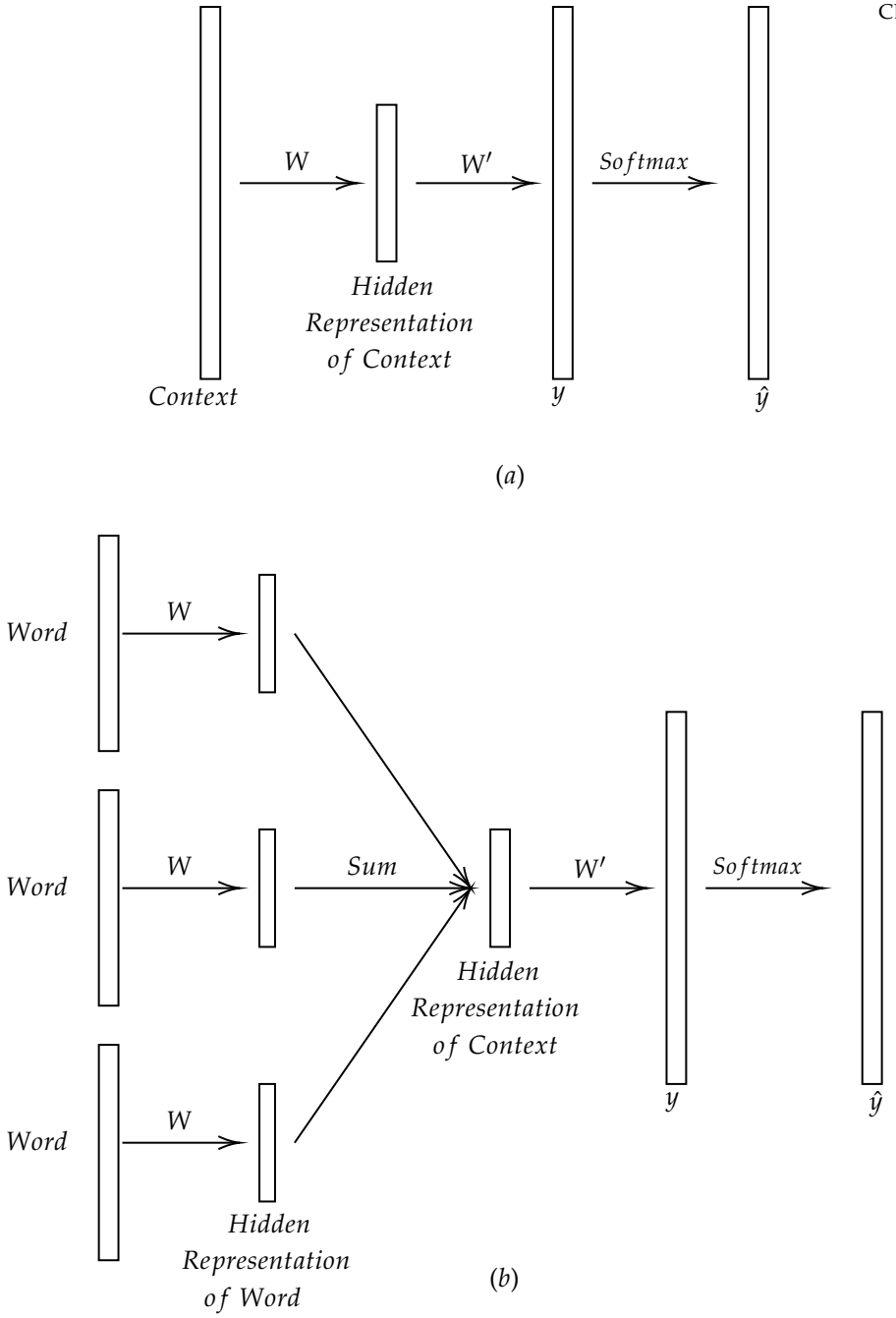


Figure 4.1: Equivalent expression of CBOW

4.4.1 *Interpretation on a two word window*

Assume we have a two word window. We use a data point (w_α, w_β). Note that:

$$\mathcal{L}(w_\beta) = -\log \hat{y}_\beta = -\log\left(\frac{e^{y_\beta}}{\sum_r e^{y_r}}\right)$$

We do back propagation and calculate the gradients:

$$\frac{\partial \mathcal{L}}{\partial y_i} = \hat{y}_i - \delta(i, \beta)$$

Denote this as E_i and define $E = [E_1, E_2 \dots E_n]$. Note $E = \hat{y}-$Observed y

$$\frac{\partial \mathcal{L}}{\partial h_i} = \sum_j \frac{\partial \mathcal{L}}{\partial y_j}\frac{\partial y_j}{\partial h_i} = \sum_j E_j \frac{\partial \sum_k h_k W'_{k,j}}{\partial h_i} = \sum_j E_j W'_{ij}$$

Therefore the gradient with respect to h is

$$\nabla \mathcal{L} = W'^T E$$

This gives the update rules for h as

$$h := h - \eta \nabla \mathcal{L} = h - \eta W \hat{y} + \eta W \text{ Observed } y$$

Assume initial entries of W to be small. Then and therefore, entries of h_{Cat} and h_{Dog} will have similar starting values: almost 0. Look at the following sentences:

Cat sleeps
Dog sleeps

Note in both those cases, Observed y will be same and therefore h_{cat} and h_{dog} will be adjusted in the same direction when they appear in similar context. This idea makes sure that similar words will have similar cosine similarity due to a form of transitive action.

4.5 *Skip-Gram*

Skip-gram in a sense is CBOW trained in opposite direction. In this case we use a word to classify what context it comes from. The structure of the model is almost same as before.

1. **Input :**For word w_i, the input is $v = e_i$
2. **Hidden Layer 1**: This is obtained by multiplying v by a matrix W. We denote:

$$h = Wv$$

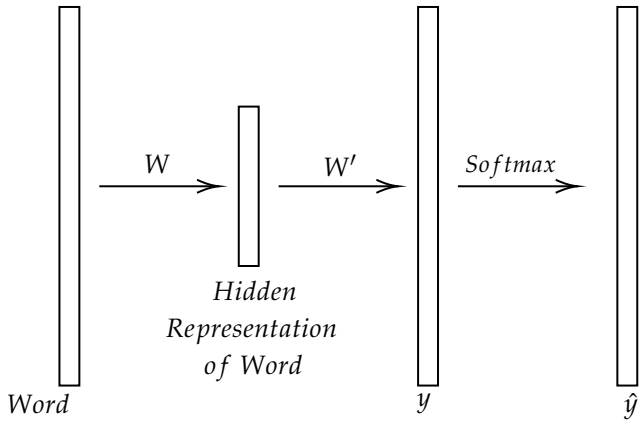


Figure 4.2: Schematic diagram of Skip gram

3. **Hidden Layer 2**: We use another matrix W' and get

$$y = W'h$$

4. **Hidden Layer 3**: This is just a soft max to normalize things:

$$\hat{y} = \text{Softmax}(y)$$

The target output is the one-hot representation of the context. If we have a window size of n our loss function, cross-entropy is the sum of $n-1$ terms. For example, if the data point has context $w_\alpha, w_\beta, w_\gamma, w_\delta$ then the loss function will be:

$$\mathcal{L} = -\log \hat{y}_\alpha - \log \hat{y}_\beta - \log \hat{y}_\gamma - \log \hat{y}_\delta$$

Note that gradients are summed. For each term, the update to h is the same as the one for CBOW. We again note that only the i^{th} column of W is updated: therefore we need not keep track of the rest of the weights. The interpretation of CBOW is carried over, this time with multi-word windows. For the sentences:

My pet Cat sleeps
My pet Dog sleeps

Since both gives the same Observed y, h_{Cat} and h_{Dog} are adjusted in same direction, making them similar.

4.6 *Negative sampling*

To improve performance, we might further want to make sure that words which are not similar are further apart. This gives rise to the idea of negative sampling. We make randomized context-word pairs which don't occur in the corpus. Therefore, we have two sets: $D =$

$\{(S,W) \in Corpus\}, D' = \{(S,W) \notin Corpus\}$. Define a random variable Z as:

$$Z = \sigma\left(\hat{y}_S^T \cdot W\right) \text{ or } \sigma\left(S^T \cdot \hat{y}_W\right)$$

We would like to maximize:

$$P(Z = 1 | (S,W) \in D)$$

$$P(Z = 0 | (S,W) \in D')$$

We use now compute the log-likelihood function as follows:

$$\begin{aligned}
&\underset{\theta}{\text{maximize}} \prod_{(w,c)\in D} p(z = 1 \mid w, c) \prod_{(w,r)\in D'} p(z = 0 \mid w, r) \\
&= \underset{\theta}{\text{maximize}} \prod_{(w,c)\in D} p(z = 1 \mid w, c) \prod_{(w,r)\in D'} (1 - p(z = 1 \mid w, r)) \\
&= \underset{\theta}{\text{maximize}} \sum_{(w,c)\in D} \log p(z = 1 \mid w, c) + \sum_{(w,r)\in D'} \log\big(1 - p(z = 1 \mid w, r)\big) \\
&= \underset{\theta}{\text{maximize}} \sum_{(w,c)\in D} \log \frac{1}{1 + e^{-v_c^T v_w}} + \sum_{(w,r)\in D'} \log \frac{1}{1 + e^{v_r^T v_w}} \\
&= \underset{\theta}{\text{maximize}} \sum_{(w,c)\in D} \log \sigma\left(v_c^T v_w\right) + \sum_{(w,r)\in D'} \log \sigma\left(-v_r^T v_w\right)
\end{aligned}$$

This leads to a better loss function. Ideally we would want k bad w, s' pairs in D' for each correct $w.s$ in D. Since the number of bad combinations is significantly more than the number of correct combinations, $k > 1$, where k is a hyperparameter to be tuned. The samples are picked from a modified unigram distribution with $p(r) \sim count(r)^p$. The original word2vec used $p = 0.75$, but it can also be trained as a hyperparameter. Unless we properly tune k and p, SVD gives better performance.

4.7 Contrastive Estimation

The problem is soft-max is expensive. So instead of using soft max, we give it a score. Assume for elements of D we get a total score of s and for elements of D' we get a total score of s'. We make our objective to maximise $max(0, s - s - m)$: We claim that if the difference is m, then we are happy with the contrast.

4.8 SVD to learn word representations

We consider two arrays: a corpus C and a vocabulary V. We assume that the context of a word is given by words at a distance d away from it. Define a co-occurance matrix $\mathcal{M}$ defined as:

$\mathcal{M}[i][j] = \{\text{Number of occurrence of } w_i \text{ and } w_j \text{ with distance at most } d\}$

Consider the SVD decomposition of $\mathcal{M}$ as $\mathcal{M} = U \sum V$. Represent the columns of U as U_is and the rows of V as V_i^T. The k^{th} rank approximation of $\mathcal{M}$ is given by:

$$\hat{\mathcal{M}}_k = \sum_{i=1}^{k} \sigma_i U_i V_i^T$$

This representation brings out latent relations between variables. Now we define the representation of the i^{th} word as the i^{th} row of $U \sum$. This reduces the size of representation while keeping the cosine similarity fixed.

4.9 GloVe Representations

We combine count based methods(like SVD) and Prediction based methods(CBOW, Skip gram). For words $V[i]$ and $V[j]$ we would like to have a representations h_i, h_j that is faithful to $\mathcal{M}$ i.e:

$$h_i \cdot h_j = \log(j|i) = \log \mathcal{M}[i][j] - \log X_i$$

$$h_j \cdot h_i = \log(i|j) = \log \mathcal{M}[j][i] - \log X_j$$

Where X_i, X_j are the counts of occurrences of X_i and X_j We combine to get:

$$h_i \cdot h_j = \mathcal{M}[i][j] - \frac{1}{2}\left(\log X_i + \log X_j\right)$$

But the problem with this is it weights all co-occurances equally. To circumvent this issue, We replace X_i, X_j as trainable weights b_i, b_j. Therefore, our loss function becomes:

$$\sum_{w_i, w_j} \left(\mathcal{M}[i][j] - b_i - b_j - h_i \cdot h_j\right))$$

4.10 Other embeddings

Suppose we have some unstructured data with some underlying notion of similarity(For example: Faces, Words, Sentences, Graphs etc.). We can generalise the above-mentioned process of embeddings to get an alternate representation of such data in a low dimensional vector space while retaining their inherent notion of similarity.

Part III

Sequence Modelling

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.

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

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.

Weichen Sun, Fei Su, and Leiquan Wang. Improving deep neural networks with multi-layer maxout networks and a novel initialization method. *Neurocomputing*, 278:34–40, 2018.