SP19 DL collaborative notes

The students of SP19 Deep Learning
Editor: Alfredo Canziani
NYU

2nd May 2019

Preface

This document aims to be a collection of lecture and laboratory notes, in an attempt to uniform the mathematical notation and gather all resources related to understand and master *deep learning* in one, single location.

These notes are divided in five parts, which will constantly link to each other. These are Part I – Theory, where the main topics will be introduced, Part II – Practice, where an intuition will be built about abstract concepts, Part III – Coding, where we'll see how to get our hands dirty with actual neural nets, on a computer, Part IV – Applications where we'll encounter real life examples and applications, and Part V – Papers summary where short summary of the papers we've discussed will be nicely collected.

Instructions

There are overall 42 hours of lectures. For each hour there is a group of people assigned to summarise what happened in class **in roughly three pages**. Each group is made of three students: two writers and a reviewer.

Writing directions

Split your writing across the five parts of this document according to where it seems fit (see Preface). Be consistent with the notation here specified.

- Use `\vect{}` and `\matr{}` to decorate vectors and matrices respectively.
- Start a new line **only and every time** you end a sentence with a period '.'; the LaTeX engine will ignore this, but `git` will love you.
- Leave an empty line to start a new paragraph, and don't use the '\\' break line (see tex.stackexchange.com/a/225925/33287).
- Add date and group's authors name **as a comment**, after every `\chapter`, `\section`, and `\subsection`.
- The transposition symbol is obtained with `^\top`. For example $(AB)^\top = B^\top A^\top$.

Feel free to create new chapters, sections, and subsections with corresponding labels `\label{chp:}`, `\label{sct:}`, and `\label{ssc:}`.

Peer reviewing within groups

Check for notation consistency, correctness, grammar, figure captioning, usage of `\cref{}` instead of `\ref{}`, `\vect{}`, and `\matr{}` to decorate vectors and matrices respectively, unnecessary use of bullet points or itemisations, missing references and use of links to papers PDF instead, usage of $\mathbb{P}$, $\mathbb{E}$, and $\mathbb{V}$ for probability, expectation, and variance respectively using `\Pr`, `\E`, and `\V`, `\mid` for the conditional vertical bar, missing backslashes for log, exp, max and any badly formatted functions, $*$ for convolutions, $\odot$ for element-wise multiplication, usage of `\caption[Short caption]{Full caption}`, use of the correct transposition symbol obtained with `^\top`, just to name a few.

Taking inspiration

You can take inspiration from the work done by the students at NYU, who collectively wrote up the lecture notes in this document last year. For example, you may reuse the following constructs,

and others, at your convenience:

$$\boldsymbol{X} = \begin{bmatrix} \text{—} \; \boldsymbol{x}^{(1)} \; \text{—} \\ \text{—} \; \boldsymbol{x}^{(2)} \; \text{—} \\ \vdots \\ \text{—} \; \boldsymbol{x}^{(m)} \; \text{—} \end{bmatrix}_{m \times n} \qquad \boldsymbol{Y} = \begin{bmatrix} \text{—} \; \boldsymbol{y}^{(1)} \; \text{—} \\ \text{—} \; \boldsymbol{y}^{(2)} \; \text{—} \\ \vdots \\ \text{—} \; \boldsymbol{y}^{(m)} \; \text{—} \end{bmatrix}_{m \times K}$$

$$\hat{\boldsymbol{A}}\boldsymbol{x} = \begin{bmatrix} \text{—} \; \hat{\boldsymbol{a}}^{(1)} \; \text{—} \\ \text{—} \; \hat{\boldsymbol{a}}^{(2)} \; \text{—} \\ \vdots \\ \text{—} \; \hat{\boldsymbol{a}}^{(m)} \; \text{—} \end{bmatrix} \begin{pmatrix} | \\ \boldsymbol{x} \\ | \end{pmatrix} = \begin{pmatrix} \hat{\boldsymbol{a}}^{(1)}\boldsymbol{x} \\ \hat{\boldsymbol{a}}^{(2)}\boldsymbol{x} \\ \vdots \\ \hat{\boldsymbol{a}}^{(m)}\boldsymbol{x} \end{pmatrix}_{m \times 1}$$

$$\boldsymbol{T}^{(1)}\boldsymbol{x} = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,k} & 0 & 0 & \dots & 0 \\ 0 & a_{1,1} & a_{1,2} & \dots & a_{1,k} & 0 & \dots & 0 \\ 0 & 0 & a_{1,1} & a_{1,2} & \dots & a_{1,k} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & 0 & a_{1,1} & a_{1,2} & \dots & a_{1,k} \end{bmatrix}_{(n-k+1) \times n} = \begin{pmatrix} \boldsymbol{a}^{(1)}\boldsymbol{x}_{1:1+k-1} \\ \boldsymbol{a}^{(1)}\boldsymbol{x}_{2:2+k-1} \\ \vdots \\ \boldsymbol{a}^{(1)}\boldsymbol{x}_{n-k+1:n} \end{pmatrix}_{(n-k+1) \times 1}$$

Have fun!

Contents

Part I

Theory

Chapter 1

Motivation

1.1 Ideas for "Generic" Feature Extraction

For much of its history, the field of Artificial Intelligence has viewed high-quality hand-crafted, task-specific features as the key to machine intelligence. In contrast, more recent years have witnessed the popularity and successes of *learned* and generic features. Preceding deep learning, many other approaches (such as SVM) for extracting generic features have been explored.

Generally speaking, feature extraction often consists of expanding the input's representational dimension such that the expanded features are more likely to be linearly separable; intuitively, points in higher dimensional space are more likely to be linearly separable due to the increase in the number of possible separating planes. To this end, feature extracting algorithms may learn some $f(x) = \sum_i w_i \phi_i(x)$, where w_i are the learned coefficients and ϕ_i are some chosen basic features. To form $\phi_i(x)$, there are (among others) five common approaches:

1. **Space tiling**: To use linear combinations of engineered features. In this method $\phi_i(x) = \phi(x, u_i)$ are used where u_i spreads over the domain. However, this method does not scale that well in the number of input dimensions d, as one would need n^d bumps/mappings/sub-functions to spread a grid of size n in each dimension.

2. **Random projections**: To compose random projection matrices and use them to get features. Randomly projected features turn out to perform well in practice.

3. **Polynomial classifier**: To use monomials as ϕ_i. This is a very common method of involving additional dimensions. Given data of dimension d, one can compose a polynomial whose terms are the products of particular dimensions. For example, for $d = 2$ (and data (x_1, x_2)), the polynomial would assume the form

$$w_0 + \sum_{a \in \mathbb{N}^*} w_{1,a} x_1^a + \sum_{b \in \mathbb{N}^*} w_{2,b} x_2^b + \sum_{c_1, c_2 \in \mathbb{N}^*} w_{3,c_1,c_2} x_1^{c_1} x_2^{c_2}.$$

   One can further limit the degree of the polynomial for computational efficiency. For instance, a degree-2 polynomial (for $d = 2$) would have the form $w_0 + w_{1,1}x_1 + w_{2,1}x_2 + w_{1,2}x_1^2 + w_{2,2}x_2^2 + w_3 x_1 x_2$.

4. **Radial Basis Functions (RBF)**: To use functions whose value depends only on the distance from the variable to a given point. For example, a commonly used function family is $\phi_i(x) = e^{-\|x-u_i\|^2}$.

5. **Kernel machines**: Based on a kernel function that satisfies the Mercer condition. More specifically, one could use $\phi_i(x) = K(x, u_i)$ where K is a continuous, symmetric, positive-definite kernel function (which indicates the matrix $[K(x_i, x_j)]_{i,j}$ is positive definite, where x_i are the sample points). In essence, taking $\phi_i(x) = K(x, x_i)$ is equivalent to a 2-layer neural network that uses tiling, that locates exactly around the training sample points (as $u_i = x_i$).

As a sidenote, it is not hard to see how a single layer classifier of the sorts discussed above may potentially require infinite dimensions.

1.2 History of Deep Learning

The seeds of deep learning date back at least to the invention of the perceptron by Frank Rosenblatt in 1957. The perceptron is a binary classifier that attempts to linearly separate the data via weights on the input features learned via supervision. The perceptron may be viewed as a single-layer neural network, and initially it seemed to be a promising statistical learning approach for classification problems (e.g., image classification). However, in 1969, Marvin Minksy and Seymour Papert showed that perceptrons cannot compute or represent some functions, for example the XOR function. This result only held for single-layer, linear perceptrons, and not in the case of non-linear multi-layer perceptrons. Despite the potential of the perceptron, the above result discouraged further research into the method. Thus, the field as it was known died and its place in research was taken by the related (or rebranded) fields of statistical learning theory and adaptive filters. These would continue to make further developments in the following years.

In the early 1980s, several exciting results re-popularized neural networks for some time. In 1982, John Hopfield discovered interesting mathematical connections between perceptrons (more generally neural networks) and concepts in physics. Soon after, in 1983, Geoffrey Hinton developed the Boltzmann machine, publishing the method in AAAI, a top tier conference in AI at the time, in a paper titled "Optimal Perceptual Inference." The Boltzmann machine was one of the first methods that could train a *multi-layer* perceptron. Later, in 1985, Rumelhart, Hinton, Lecun, and several others discovered the backpropagation algorithm for training multi-layer perceptrons. Backpropagation is conceptually simple: repeatedly apply the chain rule starting from the gradients of the last layer of the perceptron until the first layer. Interestingly, this algorithm had not been discovered before this time, given that previous neural networks had largely been formulated with binary, thresholding activations, which are non-differentiable. However, these activations could be swapped for other differentiable activations, such as sigmoid functions (a kind of soft thresholding function) or rectified linear units, in order to facilitate backpropagation. Notably, this non-linear optimization procedure faces the theoretical problem of reaching local minima. However, Hinton showed it was possible to successfully train neural nets anyways – that in practice, the solutions found by backpropagation perform reasonably well.

Despite these successes, other models (such as the Support Vector Machine) in the 1990s and 2000s overtook neural networks in popularity. It was not until the early 2010s that deep learning became popular yet again, with breakthrough successes in speech recognition and image classification.

1.3 Hierarchical Representation

Deep learning has a hierarchical nature. From simpler structures we derive more complex ones. In a neural network, the layers are essentially stages of non-linear feature transformations. This inherent hierarchy of representations can also be found on the Mammalian Visual Cortex, where the different sections of the Cortex understand increasingly complex levels of information about a visual input (as can be seen in figure 1.1).

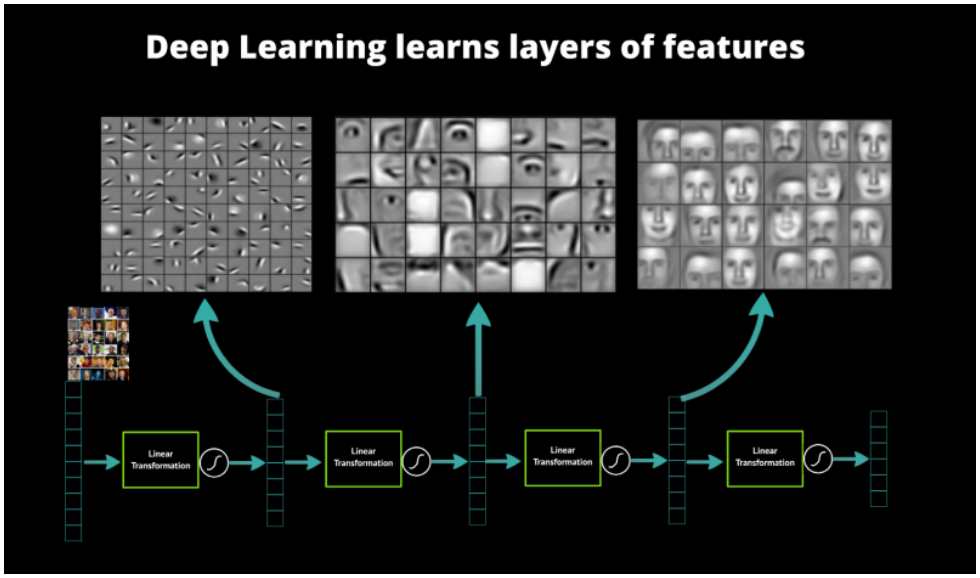


Figure 1.1: In the figure above we see how the layers closer to the input of the network are more abstract, and that those closer to the final layer are most concrete and interpretable.

In a neural network, we feed the output of one layer onto the input of the next. Layers are composed of linear transformations followed by non-linear activation functions. Altogether, this system allows the network to learn the non-linear features of interest.

Hierarchical feature representations are intuitive for many domains, as our world is compositional (the whole is often a function of its parts). For example, consider the following domains:

- **Image recognition**: Pixels compose edges, which compose "textons" (i.e., multi-edge shapes), which compose motifs, which compose parts, which finally compose whole objects. Algorithms may learn to understand what an object looks like by learning a function mapping from pixels to a slightly higher level representation such as edges, then to textons, etc., until a high enough level representation is formed to represent whole objects. Indeed, there is evidence from neuroscience that the visual cortex represents objects in such a hierarchical manner in animals.

- **Language**: Language is inherently compositional as well. The meaning of a book is a composition of the meanings of its chapters, then paragraphs, then sentences, then phrases, then words (then even characters). While not all language is compositional (a "heavy accent"

has nothing to do with weight), it often is, as language is how humans represent the world which is often compositional.

- **Speech**: In speech, a sample composes a band, which composes a sound, which compose phones, then phonemes, then whole words and sentences. The capacity for each component of this hierarchy in speech is what enables hierarchical models to represent very high-level features of sound by simply forming consecutively higher representations over what is ultimately a series of bytes.

Chapter 2

Hierarchical Representation

2.1 The World as a Hierarchy

The world is inherently compositional and hierarchical: smaller pieces combine to form larger objects. Humans interpret the world as a hierarchy; even the visual cortex in mammals is hierarchical in nature.

The goal of deep learning is to have a machine correctly extract hierarchical representations. Ideally, a deep neural network should detect features at one level, then detect combinations of those features at the next level. It is important to note that not every combination of features at one level exists in the next.

2.1.1 Images

For example, from a patch of pixels, we want to detect edges (usually by an abrupt change in color of adjacent pixels). From edges we can discern textons (e.g. corners, crosses). From textons we can detect motifs, then parts of objects, and finally those parts can be pieced together to detect objects within the image.

Geometrically, if we take all 5×5 patches of pixels in an image, we will get a collection of 25-dimensional vectors. These vectors, however, would likely comprise a small (low-dimensional) part of $\mathbb{R}^{25}$.

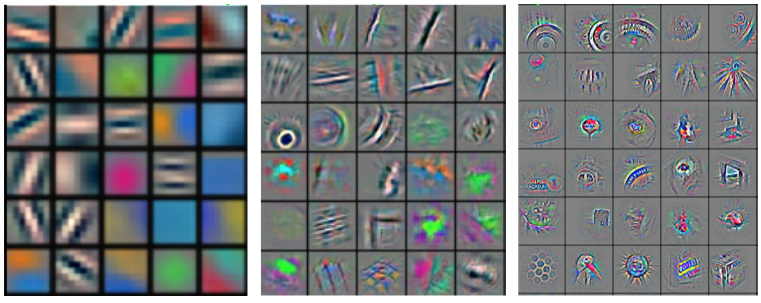


Figure 2.1: Hierarchy from a convolutional neural network

Figure 2.1 shows an example from a convolutional neural network. The left pane shows detected edges, color patches, and gradients. The middle pane has pieced those attributes together to detect textures and shapes, such as round shapes and corners. Finally, the right pane contains discernible parts of objects.

2.1.2 Text

The same idea can be applied to textual analysis: combinations of characters become words, combinations of words make word groups, which assemble to make clauses, which can be grouped to make sentences, and finally a collection of sentences create a story.

Again, not every combination of features at one level becomes significant in the next, e.g. not every combination of words forms a valid sentence.

2.1.3 Speech

An audio sample is just a single number, but the frequency content of a waveform can be represented by a feature vector, and those waveforms can be pieced together to form sounds, which can then be pieced together to form syllables, etc. Ultimately phones and phonemes are formed and pieced into words.

Chapter 3

Nonlinear Dimensionality Expansion

3.1 Motivation

A network is "deep" if it has more than one stage of non-linear feature transformation. A natural question arises: why are deep networks necessary? Theoretically, kernel machines can approximate any function with as much precision as desired. However, that might be computationally expensive—too expensive to achieve anything in practice.

3.1.1 Cover's Theorem

One way to rectify this problem is to bring the sample data into a higher dimension. Thomas Cover's theorem (1965) formalized this argument.

Cover's Theorem: say you have a linear classifier in N dimensional space with P sample points, each randomly labeled with one of two class labels. Then figure 3.1 roughly illustrates the probability that these points are linearly separable.

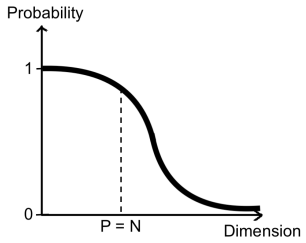


Figure 3.1: Cover's Theorem: probability of being separable vs. dimension

Ergo it makes sense to expand dimensionality of a representation because the data is more likely to be separable in a higher dimensional space. One caveat: the expansion must be *nonlinear*. That brings us to deep networks, which by definition are networks with more than one nonlinear stage.

3.1.2 The Manifold Hypothesis

Before we discuss how to expand dimensionality in a nonlinear way, we should address one concern: will working in a higher introduce an intractability problem? The manifold hypothesis suggests that it will not pose a problem. The manifold hypothesis postulates that natural data in high-dimensional space generally has a low-dimensional structure (see chapter 22 for further discussion). The shape of that low-dimensional space is dictated by the intrinsic factors of variation in the data, and our ideal feature extractor would extract those factors of variation.

3.2 How to Expand Dimensionality Nonlinearly

As described in the previous section, Cover's theorem and the manifold hypothesis urge us to expand the dimension of the representation (nonlinearly) and explore it in higher dimensional space. This is the pipeline for nonlinear dimensionality expansion:

1. Linearly expand the dimension (this can be done by multiplying by matrix with more rows than columns)
2. Apply a nonlinear transformation to each component of the vector
3. Compress the data back into a smaller dimension (linearly or with pooling)

Figure 3.2 illustrates the process of nonlinear dimensionality expansion. Between the first and second pane, the data is projected into higher-dimensional space linearly (observe the three-dimensional axes) and transformed nonlinearly. Between the second and third pane, the data is brought back down to a smaller dimension (observe the two-dimensional axes) via pooling/aggregation.

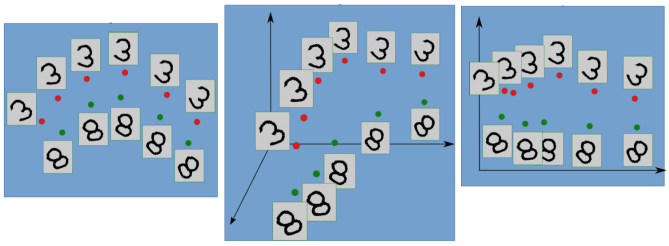


Figure 3.2: Nonlinear dimensionality expansion

3.3 In the Context of the Deep Learning System Architecture

Section 3.2 outlined the process of expanding dimensionality in a nonlinear fashion. Here are those same steps again, this time in the context of the overall architecture for a deep learning system:

1. Begin with a representation of input data
2. Normalize the input (mean = 0, standard deviation = 1)

3. Linearly expand the dimension (this can be done by multiplying by matrix with more rows than columns)
4. Apply a nonlinear transformation to each component of the vector
5. Compress the data back into a smaller dimension (linearly or with pooling)

This process can be repeated multiple times.

Chapter 4

Deep Supervised Learning: A Modular Approach

4.1 Supervised Learning

Supervised learning is the process of improving prediction accuracy of some differentiable parameterized function (**module**) G by learning from discrepancies between predicted and expected outputs. As an illustration (see figure 4.1), suppose we would like to train a function $G(x, w)$, where x is the input variable and w is a parameter called **weight**. Say we have an input-output pair $(x, y) \in \mathbb{R}^2$. The function $G(x, w)$ first takes the value of x and predicts an output $\bar{y}$. Then, the **cost function** $C(\bar{y}, y)$ measures the distance between the predicted output $\bar{y}$ and the expected output y and returns a **cost**, that is, a scalar value indicating error in prediction. Finally, the weight w will be adjusted according to the cost by using a method called **stochastic gradient descent**.

Some examples of modules can be found in section 4.4.2.

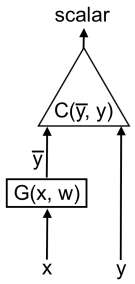


Figure 4.1: Supervised learning

4.2 Stochastic Gradient Descent

Suppose we are given a training data set consisting of P pairs of N-feature inputs and their associated outputs $(\boldsymbol{x^i}, y^i)$, where $i \in [1, P], x \in \mathbb{R}^N$, and $y \in \mathbb{R}$. We would like to optimize the weights $\boldsymbol{w} \in \mathbb{R}^N$ of some function according to errors produced by an **objective function**:

$$L(\boldsymbol{x^i}, y^i, \boldsymbol{w}) = C(G(\boldsymbol{x^i}, \boldsymbol{w}), y^i).$$

Recall that **gradient descent** calculates the derivative of the cost function in attempt to find minimal cost. If we compute the gradient for the entire training set, it may be too computationally expensive for very large training sets. Instead, a more efficient method is adopted called **stochastic gradient descent**. This method first evaluates the gradient randomly at the i^{th} pair of input and output, then $\boldsymbol{w}$ is updated accordingly with a **learning rate** η:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \left[\frac{\partial L(\boldsymbol{x^i}, y^i, \boldsymbol{w})}{\partial \boldsymbol{w}} \right]^\top. \tag{4.1}$$

The result is a row vector consisting of N partial derivatives with respect to each component of $\boldsymbol{w}$:

$$\left[\frac{\partial L(\boldsymbol{x^i}, y^i, \boldsymbol{w})}{\partial \boldsymbol{w}} \right]^\top = \begin{bmatrix} \frac{\partial L}{\partial w_1} & \frac{\partial L}{\partial w_2} & \cdots \end{bmatrix}.$$

These partial derivatives are then evaluated using a method called **backpropagation**, which will be introduced in section 4.4.1.

4.3 Multi-layer Neural Network

The above supervised learning algorithm uses only one module to predict outputs $\bar{y}$. Often, we can build a complex learning machine by assembling modules into networks. Figure 4.2 illustrates a simple example of a multimodule system. This system has a *cascade*, or *sequential*, architecture, that is, a structure where no cycles are formed in the network and the input X[1] is passed forward in only one direction. We can use a recurrence equation to represent a system with n modules:

$$X_i = F_i(X_{i-1}, W_i)$$

for all $i \in [1, n]$. Here, each module F_i is an object containing trainable parameters W_i. The output of the $(i-1)^{\text{th}}$ module X_{i-1} is returned and stored internally. It is then taken as the input for the i^{th} module F_i and so on until the signal X_0 reaches the last module F_n and computes the predicted output X_n.

Similar to a single module system, the objective function E is the cost function:

$$E(Y, X_0, W) = C(X_n, Y)$$

which measures the distance between the predicted output X_n of and the desired output Y.

[1]We simplify notations for X, W, and Y in the rest of Chapter 4 since the following abstract analysis on the structure and the gradient of a multi-layer neural network applies to scalars, vectors, or matrices of the same dimension.

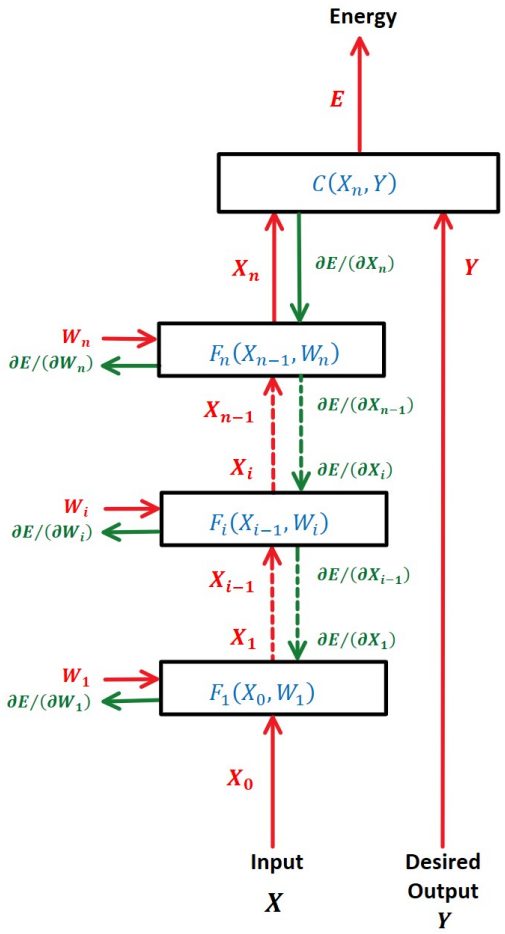


Figure 4.2: Multimodule cascade/sequential system

4.4 Backpropagation

Recall that a learning algorithm is considered "deep" if it has more than one nonlinear layer. A multimodule system with a sequential structure is often called a **multi-layer feedforward neural network**, or a **deep neural network**. See an example of defining a two-layer neural network in PyTorch in section 35.1. In PyTorch, as the signal of the initial input X_0 propagates through a sequence of approximation functions F_i, PyTorch instantaneously generates another function internally, called **backpropagation**, which propagates the cascade of modules backwards in order to compute gradients discussed in section 4.2.

4.4.1 Compute Gradient Using Backpropagation

To compute the gradient of the objective function $E(W, Y, X)$ in an n-layer neural network, first consider a module F_k in the network, where $k \in (0, n)$. The forward propagation method uses the input X_{k-1} produced by the $(k-1)^{\text{th}}$ module and computes a predicted output X_k using the weight W_k,

$$X_k = F_k(X_{k-1}, W_k).$$

Suppose we know the gradient $\frac{\partial E}{\partial X_k}$ of the cost function with respect to the output of the k^{th} module. That is, for each component of X_k, we know how much E wiggles if we wiggle that component of X_k. As mentioned in section 4.1, since the goal is to optimize weight W_k, we would like to know how much E wiggles if we wiggle each component of W_k, namely, the gradient $\frac{\partial E}{\partial W_k}$ of the cost function with respect to the weight of the k^{th} module. To do so, we can simply apply *chain rule* using $\frac{\partial E}{\partial X_k}$:

$$\frac{\partial E}{\partial W_k} = \frac{\partial E}{\partial X_k} \frac{\partial F_k(X_{k-1}, W_k)}{\partial W_k}.$$

In particular, each element (p, q) of the matrix indicates how much the p^{th} output wiggles when we wiggle the q^{th} weight:

$$\left[\frac{\partial F_k(X_{k-1}, W_k)}{\partial W_k}\right]_{pq} = \frac{[\partial F_k(X_{k-1}, W_k)]_p}{\partial [W_k]_q}.$$

Then, using the same method, we compute the gradient $\frac{\partial E}{\partial X_{k-1}}$ of the cost function with respect to the input of the k^{th} module in order to pass the signal to the $(k-1)^{\text{th}}$ module for it to repeat the above process again. The chain rule is applied:

$$\frac{\partial E}{\partial X_{k-1}} = \frac{\partial E}{\partial X_k} \frac{\partial F_k(X_{k-1}, W_k)}{\partial X_{k-1}}.$$

Note that

$$\frac{\partial F_k(X_{k-1}, W_k)}{\partial W_k}, \frac{\partial F_k(X_{k-1}, W_k)}{\partial X_{k-1}}$$

are the **Jacobian matrices** of the module F_k with respect to W_k and X_{k-1}.

The entire process of recursively calculating gradients $\frac{\partial E}{\partial X_{i-1}}$ and $\frac{\partial E}{\partial W_i}$ starting from the n^{th} module backwards to the first module in the network is called **backpropagation**, that is (see figure 4.2),

$$
\begin{aligned}
\frac{\partial E}{\partial X_n} &= \frac{\partial C(X_n, Y)}{\partial X_n} \\
\frac{\partial E}{\partial X_{n-1}} &= \frac{\partial E}{\partial X_n}\frac{\partial F_n(X_{n-1}, W_n)}{\partial X_{n-1}} \\
\frac{\partial E}{\partial W_n} &= \frac{\partial E}{\partial X_n}\frac{\partial F_n(X_{n-1}, W_n)}{\partial W_n} \\
&\vdots \\
\frac{\partial E}{\partial X_{i-1}} &= \frac{\partial E}{\partial X_i}\frac{\partial F_n(X_{i-1}, W_i)}{\partial X_{i-1}} \\
\frac{\partial E}{\partial W_i} &= \frac{\partial E}{\partial X_i}\frac{\partial F_i(X_{i-1}, W_i)}{\partial W_i} \\
&\vdots \\
\frac{\partial E}{\partial X_0} &= \frac{\partial E}{\partial X_1}\frac{\partial F_n(X_0, W_1)}{\partial X_0} \\
\frac{\partial E}{\partial W_1} &= \frac{\partial E}{\partial X_1}\frac{\partial F_1(X_0, W_1)}{\partial W_1}
\end{aligned}
$$

4.4.2 Some Module Classes

Several module classes are discussed here along with their respective gradients as they some of the most commonly used modules to build a multi-layer neural network.

Cost module. A typical cost module is a squared distance function $C = \|\bar{Y} - Y\|^2$ which measures the distance between predicted and expected outputs. It is mostly used to indicate errors in prediction.

Linear module. A linear module is a function $Y = W \cdot X$ which predicts an output Y by multiplying a Jacobian matrix W by the input X. The gradients of the cost function with respect to input and weight are

$$
\begin{aligned}
\frac{\partial C}{\partial X} &= W^\top \cdot \frac{\partial C}{\partial Y} \\
\frac{\partial C}{\partial W} &= \frac{\partial C}{\partial Y} \cdot \boldsymbol{X}^\top
\end{aligned}
$$

respectively.

ReLU module. A Rectifier Linear Unit(ReLU) is a pointwise nonlinearity $Y = ReLU(X)$ which takes an input X and returns the identity function $Y = X$ if $X \geq 0$ and 0 if $X < 0$. The gradient is

$$
\frac{\partial C}{\partial X} = \begin{cases} \frac{\partial C}{\partial Y}, & \text{if } X \geq 0 \\ 0, & \text{otherwise} \end{cases}
$$

Duplicate module. A duplicate module $Y_1 = X, Y_2 = X$ makes two copies of the input X and return two identical outputs Y_1 and Y_2. The gradient with respect to the input for this module is

$$\frac{\partial C}{\partial X} = \frac{\partial C}{\partial Y_1} + \frac{\partial C}{\partial Y_2}$$

since two costs are adjusted when we adjust the weight of the function.

Add module. An add module $Y = X_1 + X_2$ is a simple addition on two inputs X_1 and X_2. If we adjust the weight of the function, only one cost will be adjusted. Therefore, the gradients of the inputs are the same

$$\frac{\partial C}{\partial X_1} = \frac{\partial C}{\partial Y} = \frac{\partial C}{\partial X_2}$$

Max module. A max module $Y = \max(X_1, X_2)$ compares inputs X_1 and X_2 and returns the larger of the two. The gradients with respect to X_1 and X_2 are as follows:

$$\frac{\partial C}{\partial X_1} = \begin{cases} \frac{\partial C}{\partial Y}, & \text{if } X_1 > X_2 \\ 0, & \text{otherwise} \end{cases} \quad , \quad \frac{\partial C}{\partial X_2} = \begin{cases} \frac{\partial C}{\partial Y}, & \text{if } X_1 < X_2 \\ 0, & \text{otherwise} \end{cases}$$

LogSoftMax module. A LogSoftMax module given by

$$Y_i = \log\left(\frac{\exp(X_i)}{\sum_j \exp(X_j)}\right) = X_i - \log\left(\sum_j \exp(X_j)\right)$$

normalizes the given input X to a probability distribution indicating the likelihood in $[0, 1]$. The gradient is computed as follows:

$$\frac{\partial C}{\partial X_i} = \frac{\partial C}{\partial Y_i} - \frac{\exp X_i}{\sum_j \exp X_j} \frac{\partial C}{\partial Y_k}$$

Note that the gradient differs when $i = k$ and $i \neq k$.

Chapter 5

Backpropagation

5.1 Motivation

Backpropagation is the process of finding the gradients of the weights and biases of a neural network using chain rule, starting with the last layer, and moving backwards. It provides a disciplined way to train neural networks using gradient descent. At the heart of back-propagation is the use of the chain-rule for calculating gradients of composed functions. Neural networks model the composition of linear and element-wise non-linear transformations, and so the calculation of gradients to be used in model training via the chain-rule is a natural choice.

5.2 Automatic Backpropagation

Given the importance of gradients in training neural networks, frameworks, such as torch, keep track of and pre-calculate derivatives of commonly used functions or modules. The PyTorch library has both functional (stateless) and class forms of frequently used neural network modules. Class forms of these modules have constructors and forward propagation methods, and generally there is no need to compute the gradient of functions, as they are already computed and maintained in the class. Thus in module classes, state consists of two variables: one holding the result of forward propagation from the module to it successor nodes, and one holding the derivatives with respect to inputs and weights in the module to be passed backwards during back propagation.

5.3 Common Modules

Below are some of the common modules and the gradients of a cost function $E(W)$, which is parameterized by the weights, W of the neural network connections.

5.3.1 Linear Module

The linear module represents the multiplication of an input vector by a matrix of weights W and thus expresses a linear transformation of inputs. More formally, the forward propagation of this

module can be written as:

$$X_{out} = WX_{in}$$

where X_{in} is a p-dimensional vector of inputs to the module, W is a $d \times p$-dimensional matrix of weights, and X_{out} is a d-dimensional vector that is the output of the module. Now in order to back propagate through a network, we will need to calculate the Jacobian with respect to X_{in} and with respect to W. To do so, we employ the chain-rule:

$$\frac{\partial E}{\partial X_{in}} = \frac{\partial E}{\partial X_{out}} \frac{\partial X_{out}}{\partial X_{in}} = \frac{\partial E}{\partial X_{out}} W$$

Given the dimensions above, this will be a $1 \times p$ row vector of partial derivatives. To turn this vector into a column vector, we take its transpose:

$$\left(\frac{\partial E}{\partial X_{in}}\right)^\top = W^\top \left(\frac{\partial E}{\partial X_{out}}\right)^\top$$

Thus, we see that in the linear module, for the input vector X_{in}, we multiply by the weight matrix W on forward propagation and by $W^\top$ on backward propagation. To calculate the partial derivative of E w.r.t X_{in}, we need to calculate the gradient w.r.t each entry of the input vector. Consider the i-th entry of X_{out} which has the following equation:

$$X_{out,i} = \sum_j (W_{ij} * X_{in,j})$$

Here each entry of X_{in} i.e., $X_{in,j}$ influences all the entries in X_{out} through various W entries which gives rise to:

$$\frac{\partial E}{\partial X_{in,j}} = \sum_i \frac{\partial E}{\partial X_{out,i}} \frac{\partial X_{out,i}}{\partial X_{in,j}}$$

Now, to calculate the gradient with respect to the weight matrix W, we begin by looking at the partial derivative with respect to an individual entry W_{ij} of the weight matrix. The only factor influenced when $X_{out,i}$ changes w.r.t W_{ij} is $X_{in,j}$:

$$\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial X_{out,i}} \frac{\partial X_{out,i}}{\partial W_{ij}} = \frac{\partial E}{\partial X_{out,i}} X_{in,j}$$

Extrapolating from here, the Jacobian w.r.t the full weight matrix can be expressed as an outer product:

$$\left(\frac{\partial E}{\partial W}\right)^\top = \left(\frac{\partial E}{\partial X_{out}}\right)^\top X_{in}^\top$$

5.3.2 Hyperbolic Tangent - tanh

The tanh module is a popular non-linearity. Other commonly used non-linearities, such as the sigmoid function and rectified linear unit (ReLU), follow a similar pattern to tanh.

To execute non-linear transformations of the input data, in addition to the linear modules, neural nets employ point-wise non-linearities, such as the tanh function. The hyperbolic tangent is computed as follows:

$$\tanh(x) = \frac{2}{1 + \exp(-x)} - 1 = \frac{1 - \exp(-x)}{1 + \exp(-x)}$$

Given this non-linearity is applied element-wise, for forward propagation, we can look at an individual index i of the output vector X_{out}:

$$X_{out,i} = \tanh(X_{in,i} + \beta_i)$$

where β is a bias vector added to the input vector. For back propagation on the input vector and bias, we again use the chain rule:

$$\left(\frac{\partial E}{\partial X_{in}}\right)_i = \left(\frac{\partial E}{\partial X_{out}}\right)_i \tanh'(X_{in,i} + \beta)$$

$$\frac{\partial E}{\partial \beta_i} = \left(\frac{\partial E}{\partial X_{out}}\right)_i \tanh'(X_{in,i} + \beta)$$

where the derivative $\tanh'$ is:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

5.3.3 Euclidean Distance Module

Euclidean distance is a metric for determining proximity of two points in Euclidean space. This module is often used as the final layer of a neural network to compute the distance between a prediction and a ground truth label, where it is also commonly known as mean squared error (MSE).

In forward propagation, the output, scalar quantity is calculated as one half the ℓ_2 norm of the difference between the input vector X_{in} and the provided label Y, squared:

$$X_{out} = \frac{1}{2}||X_{in} - Y||^2$$

Back propagation to the input vector and label is symmetric:

$$\frac{\partial E}{\partial X_{in}} = X_{in} - Y$$

$$\frac{\partial E}{\partial Y} = Y - X_{in}$$

5.3.4 Y-connector and Addition Modules

The Y-connector and Addition modules are related, in that the two mirror each other on forward and backward propagation. The Y-connector module takes an input vector and *copies* it to two or more output vectors. The Addition module, on the other hand, takes two or more input vectors and adds them to form a single output vector.

On the backwards pass, the Y-connector needs to sum the partial derivatives with respect to each of the its output vectors in order to calculate the gradient with respect to the input vector. The Addition module need only copy the partial derivative of the output vector to each of its inputs, thus giving rise to the mirrored pattern referenced above. More formally, the Y-connector module can be formulated as follows:

- Forward: $\forall k \in [1..K] : \ X_{out}^{(k)} = X_{in}$

- Backward: $\frac{\partial E}{\partial X_{in}} = \sum_k \frac{\partial E}{\partial X_{out}^{(k)}}$

The Addition module can be expressed as:

- Forward: $X_{out} = \sum_k X_{in}^{(k)}$
- Backward: $\forall k \in [1..K]: \ \frac{\partial E}{\partial X_{in}^{(k)}} = X_{out}$

5.3.5 Switch Module

This module takes in k input vectors $X_{in}^{(k)}$ and a discrete-valued Y variable, which acts as a switch, selecting one of the k inputs to be copied to the output vector X_{out}. At first glance, this module may seem non-differentiable, but further examination reveals that indeed the gradient can be calculated by simply passing the gradient of the output to the input vector that was selected by Y and setting the gradient for all other input vectors to zero. For this module, the state variable that maintains information about forward propagation is essential to back propagation as well, since it allows for the gradient to flow backwards through the selected input.

5.3.6 SoftMax Module

The SoftMax module is an important module that is often used in multi-class supervised learning. Known also as the Boltzmann-Gibbs distribution by the Physics community, this module takes as an input a vector X_{in} and produces an output vector X_{out} that resembles a probability distribution, as each output component $X_{out,i} \in [0, 1]$ and they all sum to 1: $||X_{out}||_1 = 1$. The softmax equation for a given index j of the output vector is expressed as follows:

$$X_{out,j} = \frac{\exp(\beta X_{in,j})}{\sum_i \exp(\beta X_{in,i})}$$

Back propagation for each component i of the input vector depends on whether that component is equal to j or not. More formally:

- $i = j$: $\frac{\partial X_{out,j}}{\partial X_{in,i}} = \beta X_{out,j}(1 - X_{out,j})$
- $i \neq j$: $\frac{\partial X_{out,j}}{\partial X_{in,i}} = -\beta X_{out,i} X_{out,j}$

Chapter 6

Backprop In Practice

6.1 Use of ReLU non-linearities

Non-linear transformations are an essential part of the deep neural networks as discussed in the Chapter 3. An artificial neuron does not know the bounds of its values and can not decide whether it should fire or not, we use activation functions to check the value

$$Y = \sum(W * X_{in}) + C \tag{6.1}$$

produced by the neuron and decide if the neuron is fired or not.

(a) The step function is a threshold based activation function Activation function.

$$f(x) = \begin{cases} \text{activated}, & Y > \text{threshold} \\ \text{deactivated}, & Y < \text{threshold} \end{cases} \tag{6.2}$$

However the step function does not account for partial activation which can help learning become smoother and easier (less wiggly). It also has zero gradients, which makes it hard to train in practical setting. As an alternative, people used sigmoid or hyperbolic tangent non-linearities in neural networks, which are saturating functions.

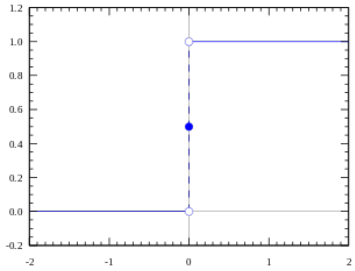


Figure 6.1: Unit Step Function

(b) The sigmoid function is a mathematical function having an "S" shaped curve (sigmoid curve).

$$f(x) = \frac{1}{1 + e^{-x}} \tag{6.3}$$

has a problem of vanishing gradients towards either end of the function causing the network to stop learning or to learn at a very slow rate.

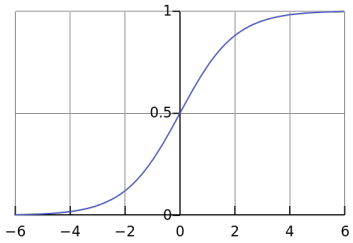


Figure 6.2: Sigmoid Function

(c) The tanh function also has vanishing gradient characteristics similar to sigmoid function,

$$f(x) = \frac{2}{(1 + e^{-2x})} - 1 \tag{6.4}$$

but it presents an improvement over the sigmoid by extending to negative values and avoiding bias in the gradients. However its gradient characteristics hinders our ability to train deeper neural network and for this reason, they fell out of favor and using ReLU non-linearities enabled us to train much deeper networks.

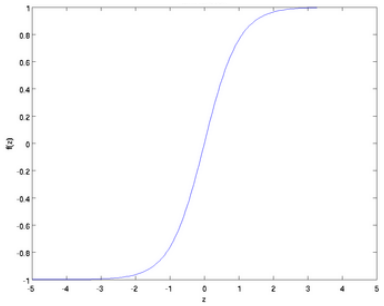


Figure 6.3: tanh Function

(d) The Rectified Linear Unit is an activation function defined as

$$f(x) = \max(0, x) \tag{6.5}$$

where x is the input to a neuron giving an output x if x is positive and 0 otherwise. This promotes a reduced likelihood of encountering vanishing gradient problems. When $a > 0$, the gradient has a constant value. In contrast, the gradient of sigmoids becomes increasingly

small as the absolute value of x increases. The constant gradient of ReLUs results in faster learning.

ReLU also helps with increasing the sparsity of the network which arises when $a \leq 0$. As the number of such units increase in a layer, it leads to a sparse representation. Sigmoids on the other hand are always likely to generate some non-zero value resulting in dense representations. The range of values can also be infinite.

ReLU is analogous to half-wave rectification in electrical engineering and has been the most popular activation function to this day.

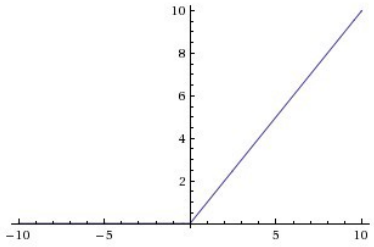


Figure 6.4: ReLU Function

Use of ReLU non-linearities

1. ReLU nonlinearities learns much faster than the standard sigmoid function and improves our ability to train deeper neural networks.
2. Also helps with increasing the sparsity of network as the neuron is switched off for $a \leq 0$

6.2 Use cross-entropy loss for classification

When we work on classification problems we use cross-entropy loss which measures performance of a classification model whose output is a probability value. The cross entropy loss increases as the predicted probability moves away from the actual label. Log loss penalizes all errors however gives higher penalty to the predictions that are confident and wrong. It is defined as,

$$H(y, p) = -\sum_{i} y_i \log(p_i) \tag{6.6}$$

we calculate a separate loss for each class label per observation and sum the result. Cross entropy measure is a widely used alternative of squared error.

Log softmax is a special case of cross entropy loss where only one value in the output array is 1 and the others are 0. The more general form calculates the cross-entropy between two discrete distributions which log-softmax doesn't compute. Softmax is often the best option to use for classification problems.

Use cross-entropy loss for classification

1. Cross entropy loss increases as the predicted probability moves away from the actual label.
2. Log softmax is a special case of cross entropy loss.

6.3 Use Stochastic Gradient on Minibatches

The standard gradient descent algorithm updates the model parameters θ in the following way:

$$\theta = \theta - \alpha \bigtriangledown \theta E[J(\theta)] \tag{6.7}$$

where the expectation in the above equation is approximated by evaluating the cost and gradient over the full training set. Stochastic Gradient Descent (SGD) simply does away with the expectation in the update and computes the gradient of the parameters using only a single or a few training examples. An estimate of the true gradient is computed based on the error E^t of that example, and the weights are updated in the following way:

$$W(t+1) = W(t) - \eta \frac{\delta E^t}{\delta W} \tag{6.8}$$

Generally each parameter update in SGD is computed w.r.t a few training examples or a minibatch as opposed to a single example.

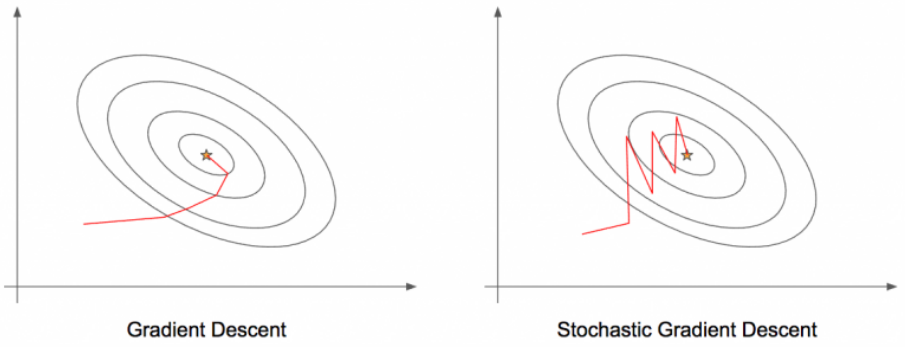


Figure 6.5: Gradient Descent vs. Stochastic Gradient Descent

We do forward and backward propagation on a small group of samples, compute their average gradients which are then used to update the model parameters. We then take the next group of samples and repeat this process. The two limit cases are when the size of the batch is one or the size of the batch is the entire training set. It is inefficient to use the entire training set to compute the gradient as there might be many redundant samples in the training set. For instance, if a training set contains one million samples which are just repetition of ten-thousand images hundred times, then by using the full batch gradient-descent we will be using hundred times the number of resources computing the gradient than we should.

We do not have such training sets in real life. However, we often have a lot of samples that are

very redundant. For instance, while doing character recognition on MNIST dataset, we will have a lot of ones very very similar to each other (with very minor variations). Similarly, ImageNet has many pictures where samples within the same category are very similar to each other. On large datasets, SGD can converge faster than batch training because it performs updates more frequently. We can get away with this because the data often contains redundant information, so the gradient can be reasonably approximated without using the full dataset. If there were no redundancy we wouldn't be able to learn any generalizable information.

However, in the presence of redundancy if we calculate the gradient on the entire training set we will be doing more computation than necessary. There has been some research done on the optimal size of a batch. There is an argument that small batch sizes put a lot of noise in the gradients, which in turn changes the weights more and produces a regularization effect that leads to better generalization error.

Another factor in determining the batch size is how well we can saturate the current hardware. Generally, the calculation in a neural net is not limited by the speed of the processor but is limited by how fast you can shuffle data in and out of the processor to and from the memory. Minibatch training can be faster than training on single data points because it can take advantage of vectorized operations to process the entire minibatch at once. How we organize the calculation within a GPU allows us to take advantage of their cache. One good way to optimize memory bandwidth is to use matrix multiplication at the lowest level (matrix multiplication is an operation that has a large number of operations per memory access thus we organize data in such a way that for every piece of data we load from memory we do a lot of operations with them).

In general, we use the smallest batch size that will saturate the hardware. Another factor that comes into consideration is parallelization. The simplest way to parallelize the training of a neural net is to take a large batch, divide it into chunks and give each chunk to a separate GPU. Their gradients will then be averaged and used for updating the model parameters, which are then distributed to each of the workers (synchronous distributed SGD).

Stochastic Learning with minibatches

1. Stochastic learning is usually much faster and uses lesser resources than batch learning.
2. Stochastic learning often results in better solutions than batch learning that produce less generalization error
3. As a rule of thumb, we can use the samllest batch size that will saturate the hardware.

6.4 Shuffle training samples

The optimum way of using the training data is by shuffling it in a way that each minibatch would have as much diversity as possible. If we were to be given a training set where the first six thousand examples are all zeros, next six thousand examples are all ones, followed by twos, threes, etc. the neural network will learn extremely slowly. This is because the model can completely ignore all the inputs and update its biases to output a given class. In this example, the model might ignore all the inputs first and learn to output zero by adjusting its biases. Then it will do the same for ones etc. It can be seen that this process would not update the input weights very often. It is, therefore, best to create diverse mini batches with our shuffling procedure. One way of accomplishing this would

be taking an equal number of samples for each class while constructing the mini-batches. There is some research suggesting that the optimal size of a batch is somewhat related to the number of categories for classification problems.

One other concern is that if a type of input is underrepresented in the training data, the model would not be incentivized to learn them well. Often we have different categories that have different frequencies in the dataset. For instance, if we train the facial recognition model where the training data has a racial distribution that reflects the racial distribution of the United States, it would result in some races to be underrepresented. This would hinder the model's performance on these types of inputs as it will focus on learning the dominant types of inputs. Eventually, it will not learn the relevant properties/features to represent those underrepresented races and would not work well for such samples.

If there is an unbalanced set of classes, one strategy that people sometimes use is downsampling. This should be avoided and we must try to make use of all available data. One way of doing this would be by picking a category at random first, and then taking a sample within that category. One can generate a counter for each class, which would denote the number of samples that were taken from that class so far. This counter would be incremented for each sample taken from that category. Once we reach the end of a given category, we would shuffle its samples, set its counter to zero and proceed further. This would equalize the frequency in a simple way while still making us able to use all the available data. However, it will also cause the model to assign similar probabilities to all categories however rare they may be. To overcome this, we can do one more pass using the real distribution of the data and re-train with real frequencies which will affect the last few layers, changing their bias and adjusting to accommodate for the real percentages. This is similar to what softmax does. One other strategy that one can encounter is modifying the loss function to have different weights for samples from different categories. However this will result in using a bigger step size for smaller underrepresented classes (that have larger multipliers) which tends to break the dynamics of the stochastic gradient descent.

Shuffling Training Samples

1. Shuffle training samples so that the samples in a batch belong to different classes.
2. Present input variables that produce a large error more frequently than examples that produce a small error. Then do one more pass with real frequencies after training the model with equalized frequencies to accommodate for real data distribution.

6.5 Normalize Input Variable

It is important for inputs to be normalized (having mean zero and standard deviation of one) so to make sure that the optimization well behaves and converges at a fast rate. A shift of the average input away from zero creates a bias in the updates in a particular direction, which slows down learning. Because of this reason it is advisable to have a zero mean over the training set. Having nonzero means, create large eigenvalues. This causes the cost surface to be steep in some directions and shallow in others, which in turn would slows the convergence.

Another factor influencing the convergence rate is the scaling of the input variables such that they

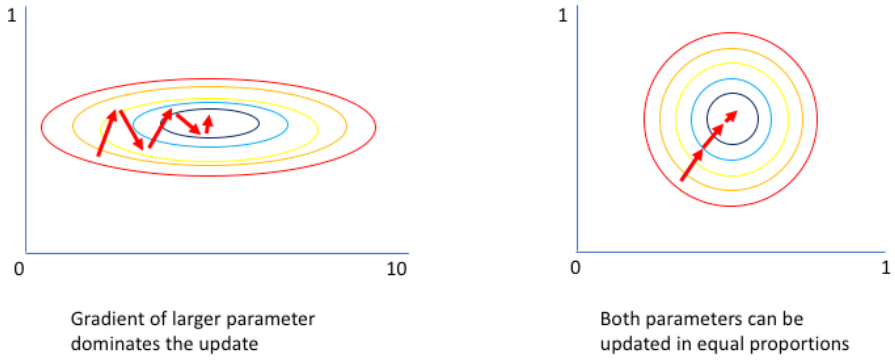


Figure 6.6: Normalizing inputs to a standard scale

have the same covariance, as inputs that have large variations in spread along different directions of the input space do also slow down the learning. This speeds up learning as it helps to balance out the rate at which the weights connected to input nodes learn. By normalizing each layer, we are introducing a level of orthogonality between layers - which generally makes it easier for the learning process.

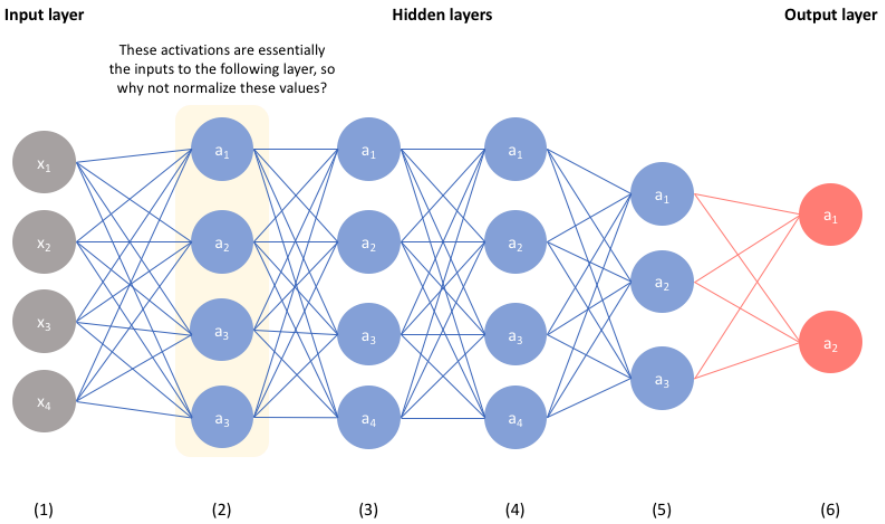


Figure 6.7: Normalization on each layer

There is a module called batchnorm which is used for the same and is placed before or after ReLU or both for this purpose. It takes the sample of the batch and normalizes input values across the minibatch. In addition to batchnorm, there are alternative techniques - featurenorm and goodnorm - which address this problem. Feature norm normalizes across the entire feature and goodnorm normalizes across the entire array. While training neural nets with gradient-based procedures, we adjust every weight according to its gradient under the assumption that other weights remain

constant. However, they change every iteration in practice. Changes in weights in layer $k-1$ can result in a change of inputs to layer k. After applying batch norm to the layers, we make sure that the inputs to every layer are close to the standard normal distribution easing dependencies between layers. To increase the stability of a neural network, batch normalization normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. Given a vector of linear combinations from the previous layer $z[l]$ for each observation i in a dataset, we can calculate the mean and variance as:

$$\mu = \frac{1}{m}\sum_i z_i^{[l]} \tag{6.9}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z_i^{[l]} - \mu)^2 \tag{6.10}$$

Using these values, we can normalize the vectors $z[l]$ as follows.

$$Z_norm^(i) = \frac{z(i) - \mu}{\sqrt{\sigma^2 + \varepsilon}} \tag{6.11}$$

We add a very small number ϵ to prevent the chance of a divide by zero error.

Batch norm is the most popular however is not that great as it depends on the batch size and we need to maintain a running average of mean and standard deviation which can not be done when running on test cases and we need an equivalent operation for that time. In goodnorm, on the other hand, we do the same operation in both testing and training.

As a result of introducing orthogonality between layers such that we avoid shifting distributions in activations as the parameters in earlier layers are updated, we can build deeper networks using normalization.

Normalize Input Variables

1. The mean of each input variable over the training set should be approaching zero.
2. Scale input variables so that their covariances are about the same.

6.6 Use a bit of L1 and L2 regularization on the weights

The use of regularization is well studied in the machine learning literature and they are used widely in different types of learning algorithms in one form or another to avoid overfitting. One form of regularization we can impose on the learning algorithm would be by putting constraints on the weights, and this is usually done by adding a term to the cost function. To that end, we can add

$$\alpha * ||w||_2^2 \quad or \quad \beta * ||w||_1 \tag{6.12}$$

terms to the cost function to impose L2 or L1 regularizations respectively. This will keep the model parameters from increasing and will increase the bias to ensure the model does not have high variance (i.e., overfitting). It will also help to remove some of the parameters if we use L1 regularization. Weight regularization also help the weights to spread around. That has some advantages for generalization as it tends to regularize the solution so that it is more general. It is

best to start regularization after a couple of epochs and crank it up as loss decreases so that the solution does not collapse in the beginning with all weights becoming zero.

Use a bit of L1 and L2 regularization on the weights

1. It keeps the model parameters from increasing and will increase the bias to ensure the model does not have high variance.
2. Best to start regularization after a couple of epochs else all weights become zero.

6.7 Use "dropout" for Regularization

Dropout is a way of regularization that makes neural nets more robust and prevents overfitting. It is a layer that randomly masks out some percentage of the units in the layer and "ignores" them during a particular forward or backward pass.

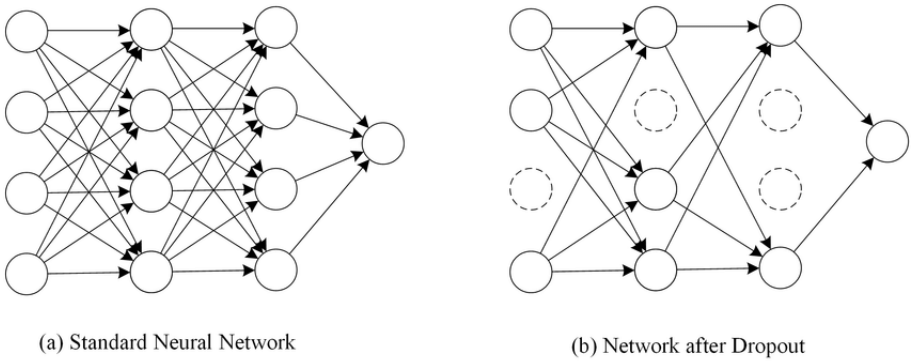


Figure 6.8: Neural network dropout

Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more responsibility for the inputs. It helps to spread the evidence to multiple features and prevents the network from giving too much importance to a particular feature. It also provides a cheap way of ensembling multiple networks. Dropout also helps reduce interdependent learning amongst the neurons and limits the networks ability to memorize very specific conditions during the training. As we turn-off some of the hidden units (with some probability), the hidden units are less inclined to learn every redundant detail of instances in training set. The subsequent layers will have access to lesser details of the input instance, they increase their efficiency learning to find patterns for unseen data, which gives a boost in generalization.

Dropout may be implemented on any or all hidden layers in the network as well as the visible or input layer. It is not used on the output layer. A new hyperparameter is introduced that specifies the probability at which outputs of the layer are dropped out, or inversely, the probability at which outputs of the layer are retained. Dropout is not used after training when making a prediction with the fit network. The weights of the network will be larger than normal because of dropout. Therefore, before finalizing the network, the weights are first scaled by the chosen dropout rate. The network can then be used as per normal to make predictions.

Use "dropout" for Regularization

1. A layer that randomly masks out some percentage of the units in the layer and "ignores" them during a pass.
2. Helps prevent the network from giving too much importance to a particular feature, and incentives to spread the evidence to multiple features

6.8 Schedule To Decrease Learning Rate

Some tricks are not well understood theoretically and are based on intuition supported by empirical evidence. One such trick is decreasing the learning rate during the training. When we observe the loss of the neural net plateaus and stops decreasing, it might be helpful to continue training with a lower learning rate.

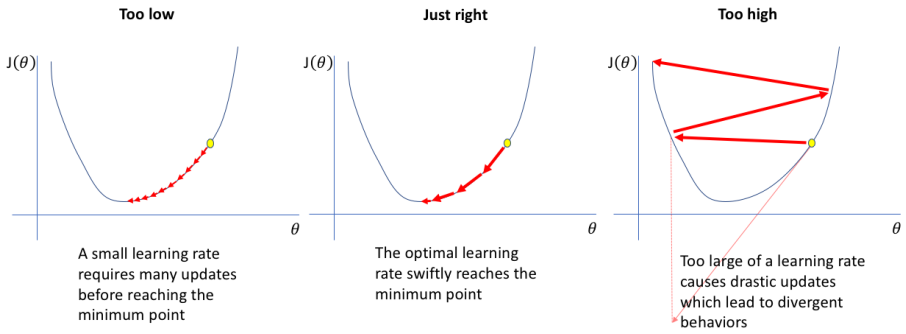


Figure 6.9: Gradient descent with small (left), optimal (middle) and large (right) learning rates

To understand why this might be helpful, imagine there are two minima (one deep and narrow and another shallow and wider) and we are optimizing the model parameters with SGD. Note that the noise in the gradients will be causing the parameters to fluctuate. The fluctuations will be smaller in the wider shallower minima than the narrow and deeper one as the weights will be bouncing off sharper in the narrower one, and they might bounce so much that the parameters will go out of the curvature. In the wider shallower minima it is very likely that the fluctuations will be smaller even though the minima is higher.

Another thing to consider is that the average loss will be higher for the steeper minima than the shallower minima. When we stop the system at one point we might get a higher loss due to bouncing. For this reason, some people say that we should not use the last value of the parameters that we obtain through SGD and instead use the average of the past few values by keeping a running average. This will help us get closer to the minimum point of a steeper minima. This, however, does not guarantee a better test loss, as the loss function that we have in the test set would be different than that of the training set. Even if it has the same shape, it might be shifted by a bit. Such a shift would affect the loss value more if we are in a steeper minima point. One intuition is that we might have a regularization effect when the system wants to choose wider minima as it has a better chance of generalizing properly. Another intuition is that regardless of which minima

(a) (b)

Figure 6.10: (a) Approaching a local minimum with changing step size (b) Slight shift in loss function resulting in a large change in loss

you are in, if you want to get to it quickly, a large learning rate might help the system approach the minima but may not let it go inside the minima because of the large fluctuation and we need to decrease the learning rate so it can settle in the minima.

Schedule To Decrease Learning Rate

1. When the loss of the neural net plateaus, it might be helpful to continue training with a lower learning rate.
2. Slight shift in loss function of test set results in a large change in loss.

One reason why neural nets disappeared in the mid-90s was because of few people who had developed tricks to make neural nets run on their system however any people who tried new neural nets would not know them and would need to figure them out themselves. That led to people just giving up.

Chapter 7

Convolutional Nets

7.1 Parameter Space Transform

Weights of a neural net do not necessarily have to be parameters. They can be outputs of other functions.

For example, in figure 7.1 the parameters of F are W but W are not the parameters that you learn. The parameters that you learn are the U but these are re-parameterized by a function $G : U \to W$ before being sent to F. This is only simple example of re-parameterizing. We could also have a function that re-paramterizes the inputs X.

If you use the functional form of PyTorch this is very easy to do. Everything works when you backpropagate. To use the above figure as an example again, when you back propagate you get the gradient of the cost with respect to the F module. Backpropagating even further gives you the Jacobian of F module times the Jacobian of the G module. The nice thing is that if you use PyTorch, all of the gradients and backpropagation is handled for you and you do not need to think about it.

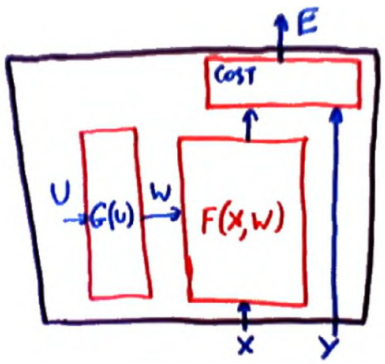


Figure 7.1: Re-parameterization of weight parameters U

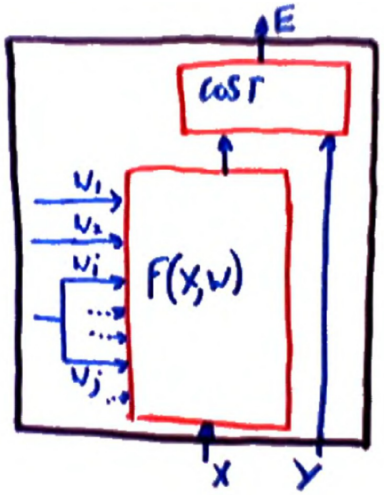


Figure 7.2: Weight sharing of weights w_i

7.2 Weight Sharing

Figure 7.2 is an example of a function G that can be used to re-parameterize the weight parameters.

In this example, G replicates a single parameter multiple times and passes the new parameters to F. When we backpropagate, the gradient with respect to that shared parameter is equal to the sum of the gradients with respect to each of the replicated parameters.

7.3 Mixture of Experts

If we have a function that works well in restricted domains of the input space but does not work globally, we want to create something that is called a Mixture of Experts. This idea of combining multiple, local optimal functions allows for very interesting architectures. This idea is actually the inspiration behind attention mechanisms that we will see later.

Imagine you have a problem with 3 distinct types of inputs. An example is recognizing faces. You might need a different model for children, male adults, and female adults. You would want 3 expert models for each of those types.

Visually, we can see that this input would benefit from 3 separate classifiers that we can combine in figure 7.3

The goal is to turn non-linear problems into a mixture of multiple linear classifiers. The mixture's sole job is to now decide where the regions are. Often times each of the expert systems is a neural net itself and is combined using a neural net to decide optimal region splitting. Softmax is often used as the final layer of a neural network for classification. These models can be trained with backpropagation just like what we have seen so far.

A good example of this is speech recognition. Imagine I want a model to work on German, French, and English. I don't want to create 3 different models. Instead I can let the model decide which language it is dealing with and then pass the sequence into different expert systems. However, in reality, the input does in fact go through all of the systems, and so we use a "Gater" module

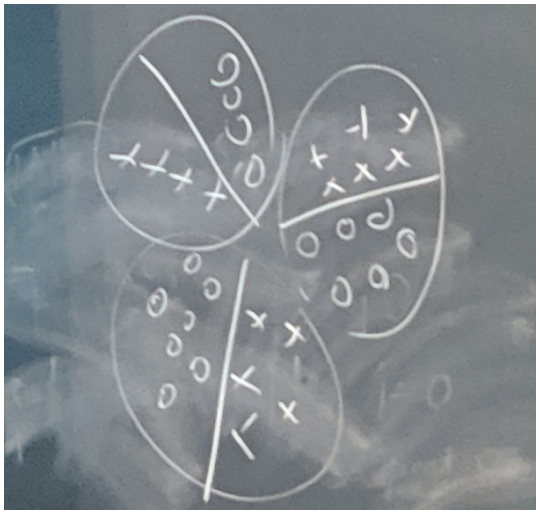


Figure 7.3: Mixture of 3 separate classifiers

to decide which expert to believe. This is all done by how our weights are tuned and how they behave with certain types of inputs. As usual, the modules are trained using backpropagation. This includes the Gater module.

7.4 Time Delayed Inputs

Let's say we want to apply a neural network to a sequence of vectors. We want to use an input that is a "time window". That is, we want to do time series predictions.

For example, you are a power company and you want to predict what the power consumption is going to be for different areas. The input is obviously historical power consumption and weather but you might also have time of day. Time-Delayed inputs are also heavily used in the financial markets.

To do this, we feed the neural network a recent window of observed values and ask it to predict the next value. During training, the output at time step t is usually the input at time step $t+1$ as in figure 7.4. The motivation for this is that like in financial markets, the thing that you want to predict has already been observed. You just want to predict when it will happen in the future. The basic technique is to take our network and "slide" it across the inputs. The same weights end up getting applied to multiple inputs in the input sequence.

An example of using time-delayed inputs is using a neural network for sound recognition on speech. Suppose you have a speech signal and you want to train a neural network to figure out what sound is being pronounced at the moment. After some pre-processing of the sound, the neural network looks at a small window of the signal. The network gives you a score for every possible sound that can exist. Slide the network over the input sequence by 10 milliseconds or so and repeat. Now for every 10 millisecond, you have a vector of scores for what sound is being pronounced.

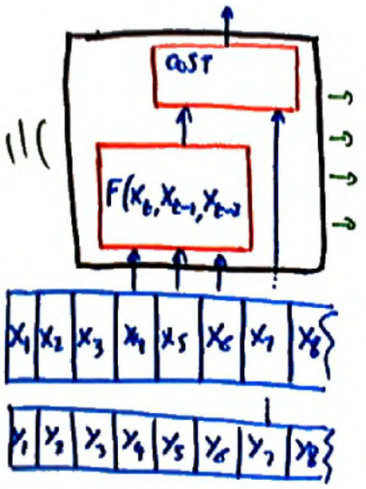


Figure 7.4: Neural network for time-delayed inputs

7.5 1D Temporal Convolutional Net

The idea we saw before of sliding the same network across different areas of the input leads to the idea of convolutions.

Image we have the 2-layer network as shown in figure 7.5. It is trying to recognize, classify or predict something from $x_{1,t}, x_{2,t}$ input sequences. The first layer is a single linear layer. Each of C_1, C_2, C_3, C_4 are shifted versions of the linear layer looking at different time steps of the inputs $x_{1,t}, x_{2,t}$. Take for instance C_1. C_1 consists of a bunch of units each looking at a 2×3 grid of the input. After the first iteration, take those same units and shift them by one time step so that the units are now looking at a 2×3 grid shifted to the right by 1. C_2 represents this newly shifted set of units. This is done until all the inputs are looked at. The second layer then does something similar. For example, in figure 7.5, $L2$ takes three different time steps of the outputs of the units in C_1, C_2, C_3, C_4 and produces an output. The important part to note here is that C_1, C_2, C_3, C_4 all contain the same units. They are just the units at different time steps.

The idea of shared weights is that many of the weights in each layer are exactly the same but used in different time steps. As you train a network like this many of the parameters are used to compute values across different time steps so when you backpropagate, you need to compute the gradient of the same weight at each time step, add them up and then update that common weight with that gradient.

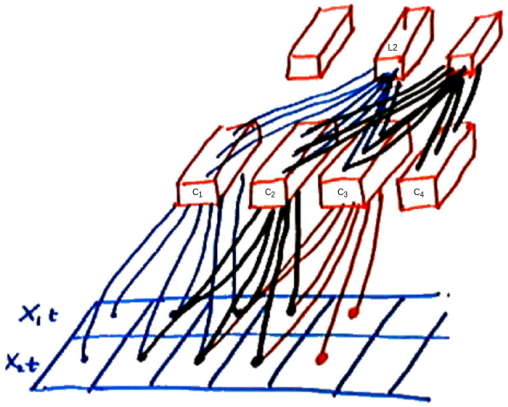
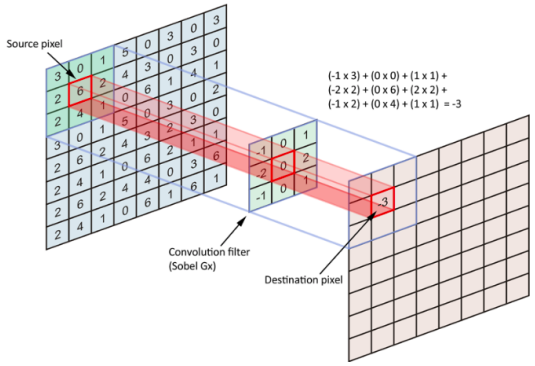


Figure 7.5: 1D (temporal) ConvNet

7.5.1 2D Temporal Convolutional Net



As you can see from the image above, that single filter has 9 weights. However, we slide that filter across the entire input grid. So, those 9 weights are shared to compute many different output values destination grid. When training a network with convolution like this, we need to keep track that each weight is being used multiple times and combine the gradients correctly.

Chapter 8

Convolutional Neural Nets

8.1 Motivation and History

Traditionally, pattern recognition would have the features extractor determined by hand then run through a classifier. More modern approaches would use unsupervised learning to get more complex features than just hand picking. Motivation for neural networks or convolutional nets is that we want the feature extractor to be learned as well. As people are expensive and gradient descent has been shown to extract features better than the best engineers.

The idea is to stack multiple layers and have them learn to extract features layer by later. The visual world is a composition of different hierarchies of features. So, ideally, a model would be able to learn simple features at the earlier layers and more complex features as more layers are introduced. This idea can be seen in Figure 1 where an image of a car has progressively more complex features extracted by different levels of a conceptual neural network.

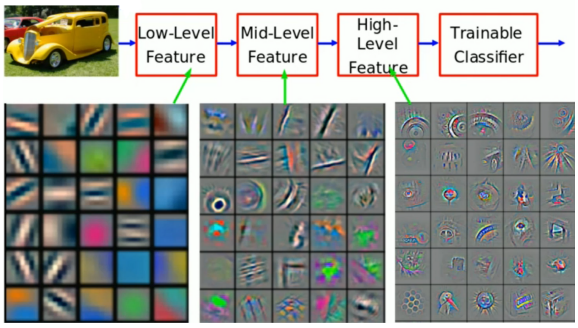


Figure 8.1: An image of a car that has different levels of features extracted by different portions of a ConvNet. Image taken from Professor LeCun's slides [Zeiler and Fergus].

This concept of learning progressively more complicated features is inspired from neurology and how the brain operates. Specifically inspired from the ventral pathway in the visual cortex. The area of the brain responsible for sight and sight processing.

In the 50s and 60s many scientists were curious with how the ventral pathway functions. It was

shown that a specific neuron would 'turn on', 'fire', or 'release neurotransmitter' when a straight edge was presented to the subject. As that edge was rotated, the original neuron would turn off and a nearby neuron would turn on. In addition, different neurons would turn on depending on the location of the edge in the input field. So, clusters of neurons would be responsible for detecting an edge, and its orientation, in a specific location of the visual field. These cells were designated with the term 'simple cells' by Hubel and Wiesel. Simple cells take input directly from the retina and detect local features.

Additionally it was noticed that some cells would turn on and stay turned on based on the input from these simple cells. So, if an edge was placed in the visual field then moved some cells would continue to be turned on. These cells were called complex cells by Hubel and Wiesel. Complex cells that 'pool' the outputs of simple cells within a retinotopic neighborhood into a more complex shape. This pooling not only creates more complicated shapes it can also removes the locality of different features.

Engineers and scientists who had the capability decided to produce models similar to the ventral pathway. Kunihiko Fukushima decided to make the neural-positron which mimicked this philosophy. Layers of these neural-positrons were created and a complicated, biologically inspired learning algorithm was used. This complicated, competitive learning algorithm was used as this was before backpropagation was created. As neurons don't have 'positive' or 'negative' weights, there were inhibitory and excitatory neurons. Lots of normalization factors as well as many tunable hyperparameters. Nonetheless Fukushima created a working model.

Inspired by Hubel, Wiesel and Fukushima - Dr. LeCun came up with convolutional neural networks in the late 80s. By taking these biologically inspired ideas and combining them with backpropagation trainable neural networks. This was done on the MNIST character dataset and showed substantial results. As this was a dark time, before the Internet, the only two datasets available with enough examples were the character dataset and the speech dataset.

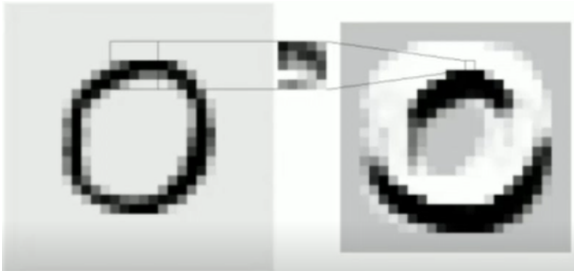


Figure 8.2: A modern rendition of one layer in a convolutional neural network. The weighted sum, aka dot product, is taken between the filter and the input image. Input image on the left. 5x5 filter, or kernel, in the middle. Output of the dot product on the right. White is a fully activated pixel. Black is a fully deactivated pixel. The output of the filter has been run through a non-linearity function (tanh) that constrained the output to be between -1 and 1. This filter is useful for detecting horizontal edges. Image taken from Professor LeCun's slides.

A rendition of one layer of a convolutional neural network can be seen in Figure 7.2. The dot product between the filter, or kernel, and the same size of the filter is conducted and that becomes a single pixel in the output image. This filter is moved over the image one, or more, pixels at a time. At each step the dot product is conducted between the input image and filter to produce a

single pixel in the output image. As pictures are worth a thousand words click here to see a filter pass over an image.

This is called a convolution. Mathematically it's actually a cross correlation as a convolution subtracts the two values instead of adds them. Regardless convolution is a better name.

8.2 Architecture

When a convolution is done over an image the size of the output image is smaller than the size of the input image. If the input image is 100x100 and the filter is 3x3 then the resulting image will be 98x98. This is because the 3x3 filter does not extend off of the input image so one pixel along the edge is lost. If this is a big deal, padding can be used around the outside of the input image. Typically the padding would be the mean of all pixel values in the image.

This shrinking of the image can be changed by using a different stride. Stride is how many pixels to move the filter in between each step.

Backpropagation is used to train this filter for detecting specific features. The values inside of the filter are considered weights to be learned. Typically initialized to 'random' or a modified random weights. The typical structure for a convolutional neural network uses multiple layers of these filters before running the output of the last convolution layer through a fully connected neural network. During training, the error is backpropagated through the fully connected neural network and then through the different filters.

This is a single filter runing over the image to produce an output. The output of each filter is typically run through a non-linear function. This non-linearity is used for the same reason non-linearity is used in a fully connected neural network. As with a fully connected neural network ReLU or tanh are popular choices of non-linearity. To allow the convolution layers to learn non-linear functions. Typically multiple filters are run concurrently over a single input image to produce multiple output images. The Conv2D module in PyTorch does all of this already. God bless PyTorch.

Typically after a convolution layer there is a pooling layer. Pooling layer shrink the image output from the convolution layer by a specified criteria. On the pooling layer, a window is run over the image. This window then takes the max, min, average or any other function between all pixels in the window. So, if the window is 5x5 and max pooling is being conducted, the maximum value of the 5x5 grid is taken as a new pixel value. The pooling layer creates a new, scaled down, image from the one provided. Yes, PyTorch also has a max pooling module. Yes, PyTorch is better than sliced bread.

The reason pooling is used is to remove the localization on features. If the pooling layer shrinks the input image by half then a feature would have to move two pixels before it is moved by one pixel in the output. If another pooling layer shrinks the input image again by half then a feature would have to move four pixels in the original image to move one pixel in the output image.

Another popular version of pooling is the p-norm. It allows the amount of pooling to be changed on the fly. The equation for the p-norm is: $\|x\|_p = (\sum_{i=1}^{n} \|x_i\|^p)^{\frac{1}{p}}$. When p is equal to one then it's just the L2 norm of the pixels. When p is equal to infinity it's just max pooling.

After a convolution layer (with non-linearity) and a pooling layer another convolution layer (with non-linearity) can be run with a 3D filter. This 3D filter takes the dot product of an area over multiple images. This allows mutliple images to be condensed together into a single output. As mentioned above, the first convolution layer could be run with mutliple filters that extract different

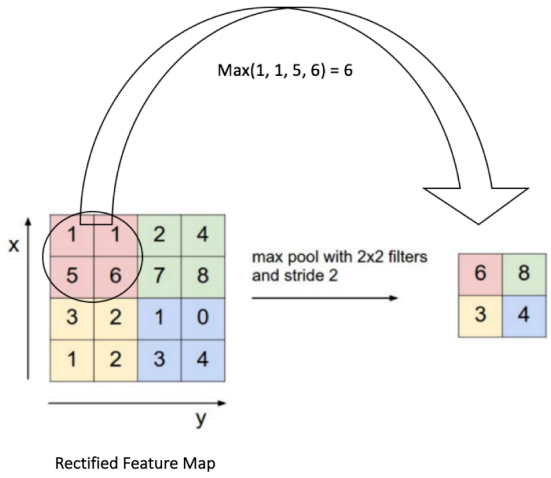


Figure 8.3: An example picture of how Max Pooling functions over an output from a Convolution Layer. Source

features from an image and output images of those features. So two filter can be used on an image to output an image of all vertical edges and all horizontal edges. The 3D filter in this case could add all the vertical and horizontal edges together to determine all crosses in the picture. PyTorch, thankfully has 3DConv layers.

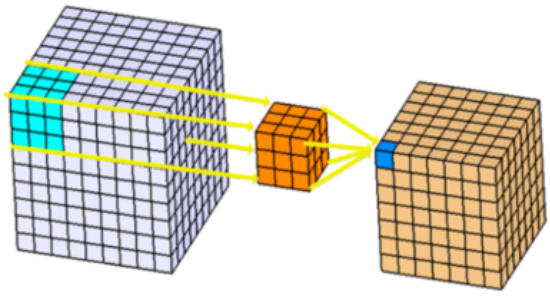


Figure 8.4: An example of a 3D filter condensing multiple images (on the left) into a new set of images on the right. If there were only 3 images on the left the resulting image would be a single image instead of a block of images. Source

Mathematically, as each filter is looking at nearby pixels, nearby pixels need to be correlated. Statistically, in natural images nearby pixels are correlated (remember cross correlation before?) and more often than not relate to a similar object. In addition, uniform areas of an image are most frequent. Then come edges between different areas of an image. Lastly comes most other features.

Empirically, to save on computation costs, small kernels are typically used. A 3x3 or 5x5 kernel requires dramatically less resources to run over an image than say a 17x17 kernel. On a 100x100 image a 3x3 kernel will need to do 98x98 matrix multiplications between two 3x3 matrices for

86,436 computations. On a 100x100 image a 5x5 kernel will need to do 96x96 matrix multiplications between two 5x5 matrices for 230,400 computations. On a 100x100 image a 17x17 kernel will need to do 84x84 matric multiplications between two 17x17 matrices for 2,039,184 (a lot) computations.

Lastly, a normalization layer may be used to correct for image whitening. A subtractive normalization layer may be used as a high pass filter. A divisive normalization layer may be used to normalize the variance of the brightness around the image. This normalization layer is wholly optional.

So generally the overall architecture of a convolutional neural network has up to four different layers: Normalization, Convolution (aka Filter Bank), Non-Linearity and Pooling. The general architecture for classification is Input Image $->$ Feature Extraction $->$ Fully Connected Neural Network $->$ Output Classification. Where the Feature Extraction can be any number of these blocks: Normalization (Optional) $->$ Convolution $->$ Non-Linear $->$ Pooling.

```
class Net(nn.Module):
    def __init__(self):
    super(Net, self).__init__()
    self.conv1 = nn.Conv2d(1, 20, 5, 1)
    self.conv2 = nn.Conv2d(20, 50, 5, 1)
    self.fc1 = nn.Linear(4*4*50, 500)
    self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4*4*50)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

This is a convolutional neural network implemented in PyTorch. This is used for digit recognition for the MNIST data set.

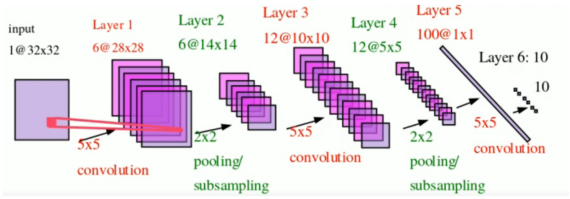


Figure 8.5: Graphical represnetation of above PyTorch code.

```
    def __init__(self):
    super(LeNet5, self).__init__()
```

```
        self.convnet = nn.Sequential(OrderedDict([
            ('c1', nn.Conv2d(1, 6, kernel_size=(5, 5))),
            ('relu1', nn.ReLU()),
            ('s2', nn.MaxPool2d(kernel_size=(2, 2), stride=2)),
            ('c3', nn.Conv2d(6, 16, kernel_size=(5, 5))),
            ('relu3', nn.ReLU()),
            ('s4', nn.MaxPool2d(kernel_size=(2, 2), stride=2)),
            ('c5', nn.Conv2d(16, 120, kernel_size=(5, 5))),
            ('relu5', nn.ReLU())
        ]))

        self.fc = nn.Sequential(OrderedDict([
            ('f6', nn.Linear(120, 84)),
            ('relu6', nn.Relu()),
            ('f7', nn.Linear(84, 10)),
            ('sig7', nn.LogSoftmax(dim=1))
        ]))

        def forward(self, img):
            output = self.convnet(img)
            output = output.view(img.size(0), -1)
            output = self.fc(output)
            return output
```

Above is (almost) the same convolutional neural network for soluving the MNIST data set problem. In this case, the network is implemented in PyTorch via the sequential object. This allows a network to be fully encompassed and easily transferrable. The main difference is that ReLU is used in this network instead of tanh. This was also take from Dr. LeCun's lecture slides.

Now, one important aspect to convolutional neural networks is that the input can be variable. So, if a convolutional neural network is trained to recognize cursive letters no modification must be done to recognize a cursive sentence. Traditionally, people would run the entire network on a section of the, now larger, input image. Then move the entire network over and run it again on the next section of the input image. With convolutional neural networks, the convolutional aspect of the network (not fully-connected portion) can be run over the whole image at once. Then the fully connected portion can be moved over the slices of the output from the convolutional portion. Running this fully-connected portion over the output layer can be though of as a convolution as well. Where the input size is the same size as the kernel size and the stride is the same size as the width of the fully connected layer. This significantly reduces computation of running a convolutional neural network over a larger image. Initially, this ability to accept varying input sizes is was called a Space Discrepency Neural Network.

There are ways to have a convolutional neural network view the overall image at once. This technique is called Convolution Et Eau?. This technique works by changing the stride size depending on the input.

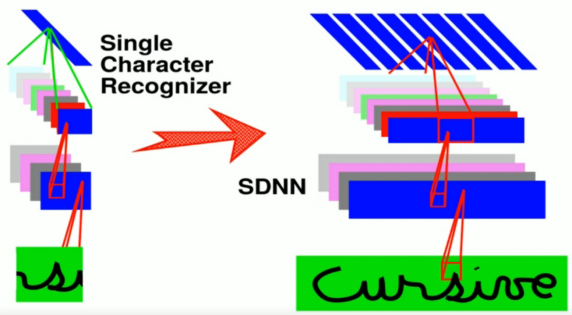


Figure 8.6: On the left is a convolutional neural network trained to recognize a single cursive character. On the right is the same network run over a larger input and the convolutional aspect fed segment by segment into the fully-connected portion. This image was taken from Dr. Lecun's slides.

8.3 Striding

Striding is a way of subsampling the input to a convolution or a pooling layer such that consecutive operations skip $S-1$ elements between their calculation (Fig. 8.7). With $S=1$ as default, there will be an output for all inputs, except for the edges. With $S=2$, there will only be an output for every other input node. The generalized equation for the resulting number of nodes is

$$M = \frac{N-K+1}{2} \tag{8.1}$$

8.4 A trou (with holes)

A trou is a similar idea to striding, but without the same cost of the resolution decrease. Instead of skipping between the kernels, a trou skips inside the kernels (Fig. 8.7). For example, with a trou of 2, one kernel would get input from every other node. This widens the receptive field of a kernel without subsampling. The generalized equation for the resulting number of nodes is

$$M = N-(K-1)S \text{ (ref. A guide to convolution arithmetic for deeplearning)} \tag{8.2}$$

where $S-1$ is the number of holes each kernel leaves while calculating the convolution. Since the resolution depends on the ratio of output nodes to the input nodes, a trou loses less resolution when compared to striding.

In the context of images we want to be able to identify the desired categories independent of location and size. CNN exploit the locality, stationarity and compositionality of the input data using sparsness, parameter sharing and hierarchical structures, respectively. Therefore, we can use these networks to not only classify but also locate and segregate data. It is doubly beneficial because convolutions are computationally cheap given that we can share parameters for a given convolution kernel (Fig. 8.7, orange selection). Since it is common there is no need to recompute it. It also means that we can only get one output for a particular kernel. So we can apply different kernels to the same input (increasing the depth of the representation) to extract different features.

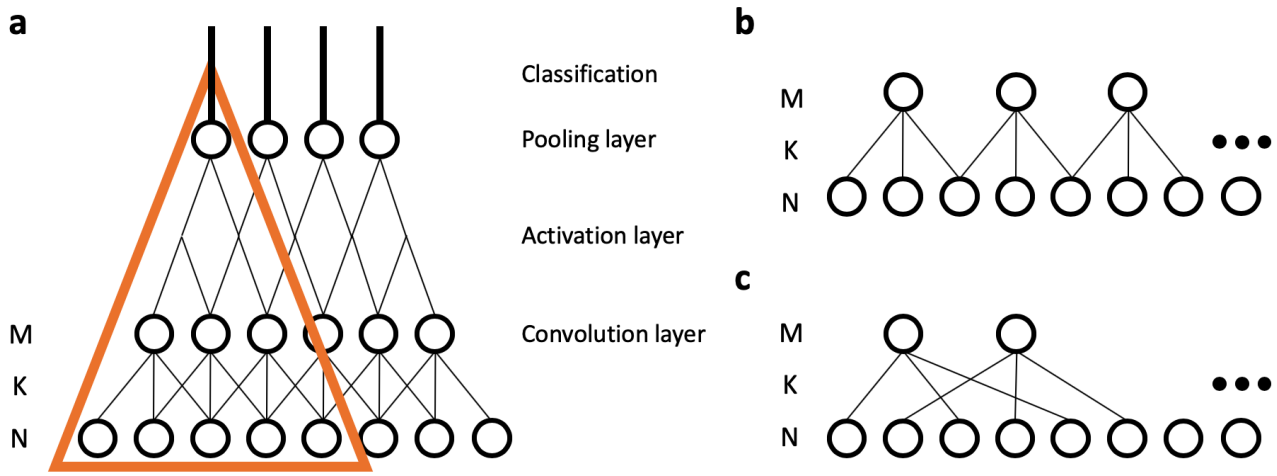


Figure 8.7: Types of convolutions in a CNN module. (a) CNN module with input layer of size N a kernel of size K and output M, followed by and activation and pooling layers. Convolution with stride 2 (b) and 'a trou' (c) diagrams.

8.5 Object detection

Let's say that we have an object detector that detects objects in an image of 6 pixels. These 6 pixels are operated on by 4 3x3 kernels to give 4 features. These four features are operated on by 2 3x3 kernels to give 2 higher level features. The final layer uses these two features to perform binary classification.

In a larger image, we can slide this object detector over the image to get a probability value of there being our desired object over those pixels. However, if we use a sliding window approach, we do a lot of extra computations because our object detector will re-compute some parts (a whole child branch leading to the output) that were already handled.

Instead we can first apply the first layer of convolutions over the entire image. From there, we can apply the second batch of convolutions and finally top it off by doing the binary classification.

If we want to emulate a fully connected layer in our object detector, we can use 1×1 convolutions.

8.6 Uses of object detection

8.6.1 Word-level training with weak supervision

This model introduced the idea of using different window widths to recognize objects (numbers) of different sizes. Using a sliding window this model can get the probability of a digit occurring over every slice of pixels.

When going over a digit, there may be multiple slices of the image where the model gives a high probability of that digit. In order not to repeat the same digit twice, this model uses a Probabilistic Finite State Machine that combines these probabilities, combining regions with high probabilities into one. This FSM can be solved using the Viterbi algorithm.

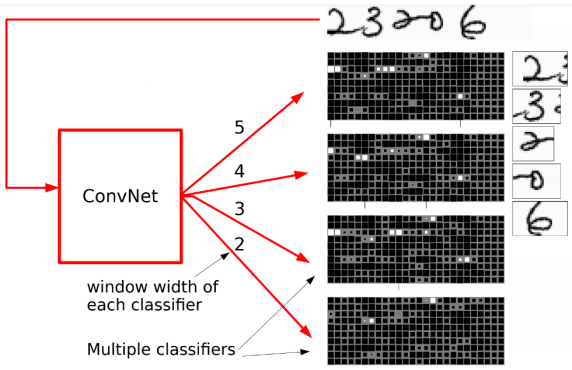


Figure 8.8: Digit recognition

8.6.2 Face Detection

At the time, the largest dataset used for face detection was in around 1000s of examples, instead of the many millions neural networks usually require. These images were of size 20×20, and the neural net used to detect faces was also quite small: there were only two convolutional layers, and a binary output.

The accuracy of this model was also not good, because there were a lot of false positives with images that looked like faces. In order to increase the accuracy, researchers needed a way of finding negative samples. The solution was to take a set of images that you knew didn't contain faces and create tons of 20×20 batches from those images. If any of those images gave a false positive, you would add that batch into the set of negative samples.

It should be noted that using white noise as negative sample is not an effective way of training, as the set of all real-life images is only a very small portion of the set of all possible pixel combinations.

This research was unfortunately largely ignored by the computer vision community.

During mid 2000s, research groups realized that detecting if there is a face and the orientation of the face at the same time led to better results compared to doing both tasks separately. This is called multi-task learning, and it forces the neural net to learn the commonality in between the two tasks. This acts as a form of regularization, leading to more generalized features for an image.

One other important outcome of this research was the use of non-maximal suppression. In non-maximal suppression (Fig. 8.9), the same image is first subsampled multiple times, creating multiple sizes of the same image.

After this sub-sampling, each kernel goes over each image, generating multiple feature maps. If there is a partial match in one of the higher resolution images, there is a good chance that there is a perfect match in one of the lower resolution images. By suppressing all of the non-maximal matches, you can find the right bounding box for an object. It also allows one kernel to detect the same feature while being size-invariant.

8.7 Semantic Segmentation

Semantic segmentation looks at the problem of identifying where exactly each object begins and end. This gives higher fidelity information when compared to using bounding boxes. Examples

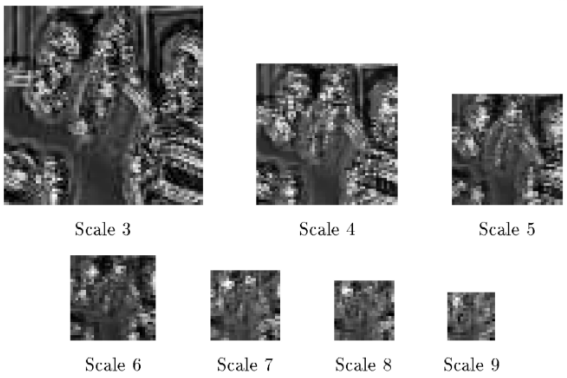


Figure 8.9: Non-maximal suppression

include medical imaging for detecting tumors (where you would want to isolate different tumors), and DARPA's LAGR program.

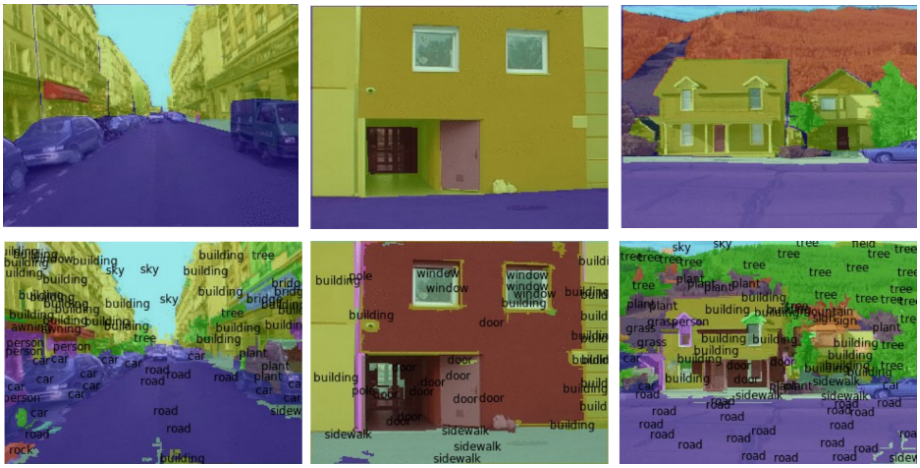


Figure 8.10: Sematic segmentation

One way of training these networks is to hand-label each and every pixel with the desired output.

For self-driving vehicles, DARPA's LAGR program used stereo vision, or stereo depth estimation, in order to estimate the relative depth of objects. Since there is a maximum range you can estimate using stereo vision, LAGR used convolutional nets to estimate longer range depth. It again used the idea of subsampling to bring down the image to multiple different resolutions before doing the convolution. This made the convolutional nets more robust (compared to stereo vision) for objects that were near or very far.

Another strategy to produce segmented images is to generate an image of patches using super-pixel to define the boundaries, and then fill each of the patches with the label that is most 'voted' by the network in that section.

To identify categories regardless of location and size, a general recipe would be:

(1) Use a CNN that takes as input the same imaged but scaled at different resolutions.

(2) To prevent false alarms, we retrain the network using a set where the category of interest is not present. We can also incorporate sets with offset focus or non-centered representations of the object.

(3) Apply non-maximum suppression. Decide the actual size of the object based on which scale of the image had the best score (or any other metric).

Chapter 9

Convolutions

The following chapter clarifies working with convolutions. Neural networks can be used to model audio, image, text, or other signals. The signals are represented as sequences of scalars. Audio is often represented as waveform heights.

9.1 Matrix Multiplication Review

Let's review multiplication between a matrix and a vector.

$$\boldsymbol{z} = \boldsymbol{A}\boldsymbol{x} = \begin{bmatrix} \text{—} & \boldsymbol{a}^{(1)} & \text{—} \\ \text{—} & \boldsymbol{a}^{(2)} & \text{—} \\ & \vdots & \\ \text{—} & \boldsymbol{a}^{(m)} & \text{—} \end{bmatrix} \begin{pmatrix} | \\ \boldsymbol{x} \\ | \end{pmatrix} = \begin{pmatrix} \boldsymbol{a}^{(1)}\boldsymbol{x} \\ \boldsymbol{a}^{(2)}\boldsymbol{x} \\ \vdots \\ \boldsymbol{a}^{(m)}\boldsymbol{x} \end{pmatrix}_{m \times 1}$$

Alternatively, we could write $\boldsymbol{z}$ as

$$z = \begin{bmatrix} z_1 \\ z_2 \\ \cdots \\ z_m \end{bmatrix} = \begin{bmatrix} \boldsymbol{a}^{(1)}\boldsymbol{x} \\ \boldsymbol{a}^{(2)}\boldsymbol{x} \\ \cdots \\ \boldsymbol{a}^{(m)}\boldsymbol{x} \end{bmatrix} = \begin{pmatrix} | \\ \boldsymbol{a}^{(1)} \\ | \end{pmatrix} \boldsymbol{x_1} + \begin{pmatrix} | \\ \boldsymbol{a}^{(2)} \\ | \end{pmatrix} \boldsymbol{x_2} + \cdots + \begin{pmatrix} | \\ \boldsymbol{a}^{(n)} \\ | \end{pmatrix} \boldsymbol{x_n}$$

What does this look like in practice? Suppose $n = 2$, we have the i-th component of $\boldsymbol{z}$

$$\begin{aligned} z_i &= \boldsymbol{a}^T\boldsymbol{x} && (9.1) \\ &= a_1x_1 + a_2x_2 && (9.2) \\ &= \|\boldsymbol{a}\| \cos\alpha \|\boldsymbol{x}\| \cos\xi + \|\boldsymbol{a}\| \sin\alpha \|\boldsymbol{x}\| \sin\xi && (9.3) \\ &= \|\boldsymbol{a}\|\|\boldsymbol{x}\|(\cos\alpha\cos\xi + sin\alpha\sin\xi) && (9.4) \\ &= \|\boldsymbol{a}\|\|\boldsymbol{x}\| \cos(\alpha - \xi) && (9.5) \end{aligned}$$

This means $\boldsymbol{z}$ represents the *alignment* between each row of A and $\boldsymbol{x}$. If we $\boldsymbol{x}$ unitary, we have the norms as 1, then $\boldsymbol{z}$ has just cosine values as entries. Here, aligning perfectly gives us the maximum value of positive 1, while going the other direction achieves the most negative value -1.

The next section talks about how this relates to convolutions.

9.2 Convolutions

Let $\boldsymbol{A}$ of dimension 3×4 be our kernel. Consider the mapping that transforms $x_{11}, x_{12}, x_{13} :\to \boldsymbol{a}^{(1)} x[1:3]$, and filling in the values as the top left 1×3 block of a new matrix. Completing this mapping, we construct the following transformation $\boldsymbol{T}$: every component $\boldsymbol{T^{(i)}}$ is multiplied by $\boldsymbol{x}$ to obtain a convolution. More generally for width k, we have

$$\boldsymbol{T}^{(1)}\boldsymbol{x} = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,k} & 0 & 0 & \dots & 0 \\ 0 & a_{1,1} & a_{1,2} & \dots & a_{1,k} & 0 & \dots & 0 \\ 0 & 0 & a_{1,1} & a_{1,2} & \dots & a_{1,k} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & 0 & a_{1,1} & a_{1,2} & \dots & a_{1,k} \end{bmatrix}_{(n-k+1)\times n} \quad \boldsymbol{x} = \begin{pmatrix} \boldsymbol{a}^{(1)}\boldsymbol{x}_{1:1+k-1} \\ \boldsymbol{a}^{(1)}\boldsymbol{x}_{2:2+k-1} \\ \vdots \\ \boldsymbol{a}^{(1)}\boldsymbol{x}_{n-k+1:n} \end{pmatrix}_{(n-k+1)\times 1}$$

$\boldsymbol{T^{(i)}}$ here is a *Toeplitz matrix*. What is the reason that $\boldsymbol{T^{(i)}}$ has 0s padded in on the top right and lower left corners?

Natural signals (not artificial or synthetic) tend to exhibit two important patterns, which are both crucial for convolutional neural networks:

Stationarity

1. Stationarity - Patterns within the signal repeat themselves in multiple places throughout the signal. The type of features encountered in the signal don't depend on the location within the signal. The statistics of one part of the signal are the same as any other part. Some sources also note that stationarity implies that "features that are useful in one region are also likely to be useful for other regions. We keep the input stationary by utilizing parameter-sharing, so $\boldsymbol{a}^{(1)}$ is used through out all rows of $\boldsymbol{T^{(i)}}$.

Locality

2. Locality - The correlation is high between nearby datapoints, but lower and lower between data points that are farther and farther away. So we don't care about the points that are far away, since they tend to be less related in natural signals.

Chapter 10

Components of a CNN

10.1 Variety of layers

A convolutional neural network may consist of various kinds of layers. Each layer is responsible for extracting relevant information and creating higher-level interpretations of the input. In a standard CNN, an input will go through a series of layers such as convolution, non-linearity, pooling, and batch normalization layers.

10.1.1 Convolution

A convolution layer is the backbone of all convolutional neural networks. The layer contains a set of learnable filters that will produce an activation map displaying how the input has responded to each filter. For instance, a filter could attempt to detect a certain kind of edge present in the input, or in higher levels of the network detect a shape. This is achieved by using small spatial filters that convolve over the height and width of the image, computing the dot product at every position. The output of each filter will produce a two-dimensional activation map which will be stacked along the depth-dimension to create the output for the layer.

10.1.2 Non-linearity

Without a non-linearity module present in the network, no matter how many layers are defined, the neural network would behave like a single-layered model. This is due to the fact that summing these layers would simply output another linear function. Additionally, with the inclusion of non-linearity modules, non-linearity is introduced into the model and thus more complex concepts can be learned. The ReLU activation function is typically used as opposed to hyperbolic tangent as it is faster to train. Despite the theoretical need for nonlinearities to be added, there is a paper that was able to train a deep model without nonlinearity layers. Interestingly enough, the floating point approximations from the calculations were enough of a nonlinearity to train the model.

10.1.3 Pooling

Downsampling the input is an important component of learning representations with a CNN. A pooling layer after a convolution layer is used to progressively downsample the spatial size of the

representation. Thus, the amount of parameters and computation in the network is reduced. This helps in preventing overfitting. Additionally, a pooling layer is essential in establishing translation invariance. An edge detected at one corner of the image will still be an edge when picked up at a different part of the image (and possibly rotated as well). Max-pooling will let the network preserve this edge despite losing its location.

In practice, using 2×2 pooling with a stride of 2 for instance, 75 percent of the information will be lost. Since much of the information is lost, it is generally a rule of thumb that the depth of the feature map is doubled using a convolutional layer beforehand. Then, the pooling is performed with only a 50 percent information loss instead.

10.1.4 Batch normalization

The batch normalization layer is typically invoked after a non-linearity.

$$\mu_B = \frac{1}{m}\sum_i^m x_i$$
$$\sigma_B = \frac{1}{m}\sum_i^m (x_i - \mu_B)^2$$
$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B + \epsilon}}$$

Batch normalization is a commonly used trick to speed up training performance in convolutional neural networks, while providing robustness to the network. A batch normalization layer normalizes the data in each batch to have zero mean and unit variance. It provides the ability to decouple subsequent layers such that the current layer does not have to keep track of the moving statistics of the previous layers. If batch normalization is not applied, every time a weight is changed in the lower layers everything propagates up and there is too much interaction between the upper and lower layers. It also provides some consistency between layers by reducing internal covariate shift (paper).

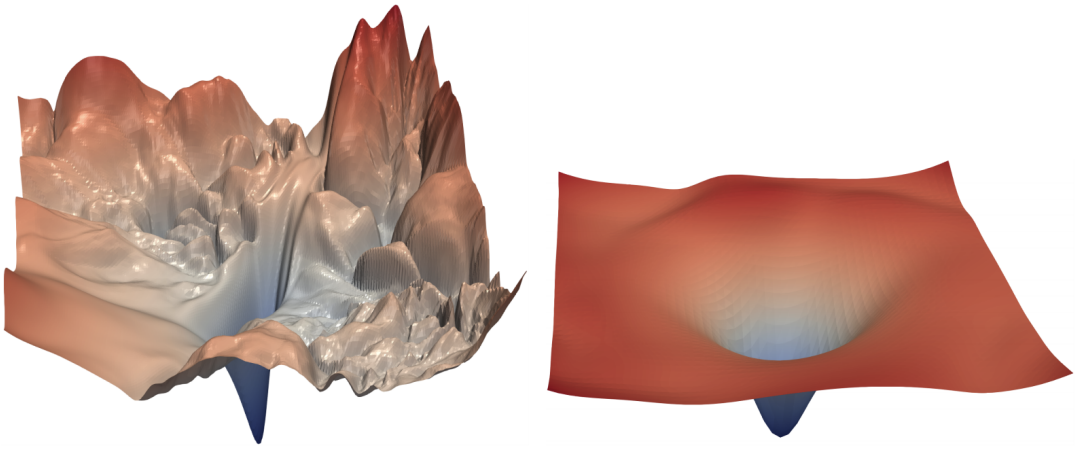
10.2 Residual Connections

An important problem with training deeper neural networks has been the loss of gradients as deeper parts of the network are approached. This is commonly referred to as the **vanishing gradient problem**. One way the deep learning community has been able to overcome this issue is by using residual connections in the neural network (paper). A residual connection (or skip connection) connects an output of a layer to another layer by jumping over layers. The weights of the skip connections are learned as the network learns. By skipping over layers, the network is able to propagate gradients and ensure better representation learning. General construction looks like:

$$y_k = h(x_k) + \mathcal{F}(x_k, \mathcal{W}_k)$$
$$x_{k+1} = f(y_k),$$

where $h(x_k)$ is the bypass short cut and $h(\_)$ is commonly taken to be identity map.

There are a decent amount of variants proposed in the research community which are based on skip connections. DenseNets (paper) has several parallel skip connections within each block.



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

Figure 10.1: The loss surfaces of ResNet-56 with/without skip connections.

Figure 1 (original paper) depicts how the use of residual connections makes the loss landscape smooth, which allows the network to converge.

10.3 Information Bottleneck

As information moves up the layers, it passes through different layers which filter out relevant features. This commonly requires altering the dimensions of the information. For example, recall that one reason pooling is used is to reduce the size of the feature maps. This dimensionality reduction plays an important role in how the network learns. The method of extracting relevant information and forgetting the rest is known as a procedure called “Information Bottleneck". There has been an increasing interest in the information theory aspect of how deep neural networks work, and how the architectures help them generalize well (paper).

Chapter 11

A digression on the Fourier Transform

11.1 Definition

The Fourier transform takes a function of time (sometimes called a signal) and decomposes it into its frequencies. The Fourier transform refers to both the operation that decomposes the function and the resulting decomposition. The domain of the Fourier transform $\hat{f}(\xi)$ is called the "frequency domain".

The Fourier transform maps a function $f : \mathbb{R} \to \mathbb{C}$ into a function $\hat{f}$ as follows:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i \xi x} dx \tag{11.1}$$

Note that this can be inverted so that given the Fourier transform $\hat{f}$ in the frequency domain we can recover the original function f.

In machine learning, we discretize the data and use the discrete Fourier transform. Given a sequence $x_0, \ldots, x_{n-1}$ we can map this to a sequence of frequencies $\hat{x}_0, \ldots, \hat{x}_{n-1}$:

$$\hat{x}_k = \sum_{j=0}^{n-1} x_j e^{2\pi i (kj/n)} = \sum_{j=0}^{n-1} x_j (\cos(2\pi kj/n) - i \sin(2\pi kj/n)) \tag{11.2}$$

The discrete Fourier transform can be calculated quickly by the Fast Fourier Transform (FFT) algorithm. There is also an analog of the Fourier transform for graphs, which is closely related to the graph Laplacian.

11.2 Connection to convolution

One nice property of the Fourier transform called the convolution theorem says that Fourier transform of the convolution of two functions is equivalent to product of the Fourier transforms of the functions in the frequency domain. A similar theorem applies to cross-correlations (which is what Convnets are doing). One nice application of this is that according to Yann sometimes cudnn

calculates convolutions by finding the FFT and performing multiplication in the frequency domain and then using the inverse Fourier transform to recover the result.

Chapter 12

Graph CNN and spectral networks

12.1 Why do CNNs work so well

CNNs have worked well on Euclidean structures (e.g. a regular grid) because of the following properties:

1. Images, videos and speeches have translation invariant properties. For example, one can consider images as sampled instances from distributions on the Euclidean space. They are stationary in that they are shift-invariant.
2. Images are scale separation
3. Features are usually localized, meaning they are much smaller than the input image
4. CNNs have a fixed number of parameters
5. With advancement in graphic gards, CNNs can be computed efficiently

However, it is not immediately obvious how to extend CNNs to graphs because they do not have these properties as in Euclidean space.

12.2 Extend CNN on Graphs

12.2.1 A bit of graph theories

1. A weighted undirected graph G with vertices $V = 1, ..., n$, edges $E \subseteq V \times V$ and edge weights $wij \geq 0$ for $(i, j) \in E$.
2. We can define functions over the vertices $L^2(V) = V \to R$, which is also a vectors $f = (f_1, ..., f_n)$.
3. We can define the unnormalized Laplacian for f as $(\triangle f)_i = \sum_{j:(i,j)\in E} w_{ij}(f_i - f_j)$ which is the difference of f and its local average.

By doing eigendecomposition of a graph Laplacian, we can obtain the its orthogonal eigenvectors as well as the corresponding non-negative eigenvalues.

12.2.2 Fourier analysis on Euclidean spaces

A function $f : [-\pi, \pi] \to R$ can be written as a Fourier series as the following formula:

$$f(x) = \sum_{k \geq 0} \langle f, e^{ikx} \rangle_{L^2([-\pi,\pi])e^{ikx}} \tag{12.1}$$

and the corresponding Fourier basis are the Laplacian eigenfunctions: $k^2 e^{ikx}$

12.2.3 Fourier analysis on graphs

A function $f : V \to R$ can be written as a Fourier series as the following formula:

$$f(x) = \sum_{k=1}^{n} \langle f, \phi_k \rangle_{L^2(V)\phi_k} \tag{12.2}$$

and the corresponding Fourier basis are the Laplacian eigenfunctions: $\lambda_k \phi_k$ with λ_k being the frequency.

12.2.4 Convolution on Euclidean space

Given two functions $f, g \in [-\pi, \pi] \to R$, their convolution can be written as

$$(f \star g)(x) = \int_{-\pi}^{\pi} f(x')g(x - x')dx' \tag{12.3}$$

12.2.5 Spectral convolution

Finally, spectral convolution can be defined by analogy to convolution on Euclidean space as

$$(f \star g)(x) = \sum k \geq 1 \langle f, \phi_k \rangle_{L^2_(v)} \phi_k \tag{12.4}$$

which is the inverse Fourier transform.

Chapter 13

Recurrent Nets

13.1 Simple Recurrent Net

RNN is designed to capture sequential information. Some of the inputs of traditional neural network are independent of each other. But in many tasks, the inputs are actually dependent of each other. In order to predict what the next word is in a sentence, you better know which words appear before it. Recurrent nets perform the same task for every element of a sequence, with the output being depended on the previous computations. RNNs can take advantage of information in arbitrarily long sequences theoretically, but in reality they are limited to capturing long-term dependencies. Here is what a typical RNN looks like:figure 13.1

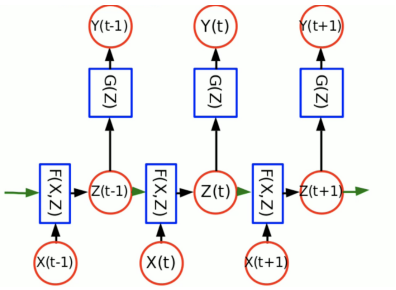


Figure 13.1: Simple Recurrent Net

- $\boldsymbol{x_t}$ is the input at time step t
- $\boldsymbol{z_t}$ is the hidden state at time t. $\boldsymbol{z_t}$ is calculated based on the input at the current step and the previous hidden state: $\boldsymbol{z_t} = F(\boldsymbol{x_t}, \boldsymbol{z_{t-1}})$
- $\boldsymbol{y_t}$ is the output at step t. $\boldsymbol{y_t} = G(\boldsymbol{z_t})$

13.2 Issue of Simple Recurrent Net

13.2.1 Backpropagation Through Time

In order to know the issue of simple RNN, we first need to know how the gradient is calculated in terms of RNN. Suppose the hidden state($\boldsymbol{z_t}$) is calculated as below:

$$\begin{aligned}\bar{\boldsymbol{z}}_{\boldsymbol{t}} &= \boldsymbol{W_X x_t} + \boldsymbol{W_Z z_{t-1}} \\ \boldsymbol{z_t} &= f(\bar{\boldsymbol{z}}_{\boldsymbol{t}}) \\ \boldsymbol{y_t} &= g(\boldsymbol{z_t})\end{aligned}$$

To measures the effect of the $(t-n)$-th input symbot x_{t-n}, where $n \leq t$, on the t-th hidden state $\boldsymbol{z_t}$ of the simple recurrent neural network, we need to calculate the derivative shown below

$$\frac{\partial \boldsymbol{z_t}}{\partial \boldsymbol{x_{t-n}}} = \frac{\partial \boldsymbol{z_t}}{\partial \boldsymbol{z_{t-n}}} \frac{\partial \boldsymbol{z_{t-n}}}{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t-n}}} \frac{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t-n}}}{\partial \boldsymbol{x_{t-n}}}$$

Among these three terms in the right hand side of the above equation, we will focus on the first term

$$\frac{\partial \boldsymbol{z_t}}{\partial \boldsymbol{z_{t-n}}} = (\underbrace{\frac{\partial \boldsymbol{z_t}}{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t}}}}_{a} \underbrace{\frac{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t}}}{\partial \boldsymbol{z_{t-1}}}}_{b})(\underbrace{\frac{\partial \boldsymbol{z_{t-1}}}{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t-1}}}}_{a} \underbrace{\frac{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t-1}}}{\partial \boldsymbol{z_{t-2}}}}_{b}) \dots (\underbrace{\frac{\partial \boldsymbol{z_{t-n+1}}}{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t-n+1}}}}_{a} \underbrace{\frac{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t-n+1}}}{\partial \boldsymbol{z_{t-n}}}}_{b}) \tag{13.1}$$

First, consider equation (13.1)(a), which is the derivative of a nonlinear activation function used in the simple recurrent neural network.

Next, we look at equation (13.1)(b). We know

$$\frac{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t}}}{\partial \boldsymbol{z_{t-1}}} = \boldsymbol{W_Z}$$

From these two, we get

$$\begin{aligned}\frac{\partial \boldsymbol{z_t}}{\partial \boldsymbol{z_{t-n}}} &= (\frac{\partial \boldsymbol{z_t}}{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t}}} \frac{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t}}}{\partial \boldsymbol{z_{t-1}}})(\frac{\partial \boldsymbol{z_{t-1}}}{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t-1}}} \frac{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t-1}}}{\partial \boldsymbol{z_{t-2}}}) \dots (\frac{\partial \boldsymbol{z_{t-n+1}}}{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t-n+1}}} \frac{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t-n+1}}}{\partial \boldsymbol{z_{t-n}}}) \\ &= (\frac{\partial \boldsymbol{z_t}}{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t}}} \boldsymbol{W_Z})(\frac{\partial \boldsymbol{z_{t-1}}}{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t-1}}} \boldsymbol{W_Z}) \dots (\frac{\partial \boldsymbol{z_{t-n+1}}}{\partial \bar{\boldsymbol{z}}_{\boldsymbol{t-n+1}}} \boldsymbol{W_Z}) \\ &= \prod_{i=t-n+1}^{t} (\frac{\partial \boldsymbol{z_i}}{\partial \bar{\boldsymbol{z}}_{\boldsymbol{i}}} \boldsymbol{W_Z})\end{aligned} \tag{13.2}$$

Suppose the recurrent activation function f is linear, equation (13.2) is reduced to

$$\frac{\partial \boldsymbol{z_t}}{\partial \boldsymbol{z_{t-n}}} = \boldsymbol{W_Z}^{n-1} \tag{13.3}$$

13.2.2 Exploding Gradients

When we do back-propagation in the recurrent neural networks, gradients of loss with respect to the weights can accumulate and result in very large gradients. These in turn cause large updates to the neural network weights.

The explosion occurs through exponential growth by repeatedly multiplying gradients through the network layers that have values larger than 1.0.

From equation (13.3), $\frac{\partial \boldsymbol{z_t}}{\partial \boldsymbol{z_{t-n}}}$ will likely explode as $n \to \text{inf}$ if $e_{max} > 1$, where e_{max} is the largest eigenvalue of $\boldsymbol{W_Z}$.

13.2.3 Vanishing Gradients

In recurrent neural networks, the gradients of loss with respect to the weights may also get smaller and smaller as we keep on moving backward in the network. In other words, the weights in the earlier layers get updated slowly compared with the neurons in the later layers. These in turn result in a difficulty for training earlier layers of recurrent neural networks.

From equation (13.3), $\left\| \frac{\partial \boldsymbol{z_t}}{\partial \boldsymbol{z_{t-n}}} \right\| \to 0$ when $e_{max} < 1$, where e_{max} is the largest eigenvalue of $\boldsymbol{W_Z}$.

13.3 Gradient clipping

Fortunately it's easy to solve the problem of exploding gradients by doing gradient clipping Pascanu et al. [2012]. First, in order to detect whether the exploding gradient arises, we could inspect the norm of the gradient of loss with respect to the parameters $\|\nabla\|$. If the gradient's norm is larger than some predefined threshold $\tau > 0$, we can renormalize the norm of the gradient to be τ.

$$\widetilde{\nabla} = \begin{cases} \tau \frac{\nabla}{\|\nabla\|} & \text{if } \|\nabla\| > \tau \\ \nabla & \text{otherwise} \end{cases}$$

13.4 Long Short-Term Memory

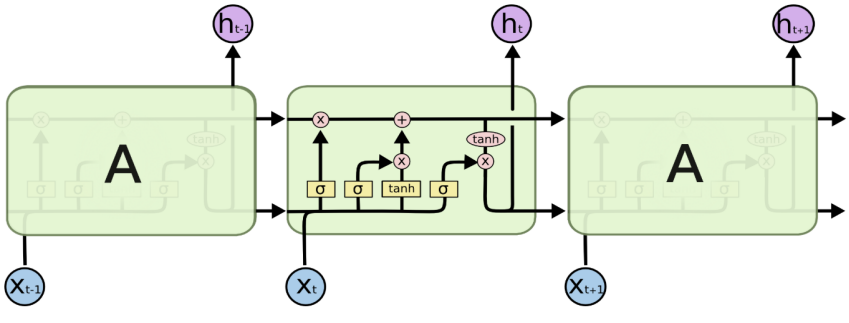


Figure 13.2: The repeating module in an LSTM contains four interacting layers. Source

Long Short Term Memory networks Hochreiter and Schmidhuber [1997] are a special kind of RNN, capable of learning long-term dependencies. They work tremendously well on a large variety of problems, and are now widely used. LSTMs are explicitly designed to avoid the long-term dependency problem. Below is how LSTM works

The forget gate in LSTM is to decide what information we're going to throw away from the cell state.

$$\boldsymbol{f_t} = \sigma(\boldsymbol{W_{fh}}\boldsymbol{h_{t-1}} + \boldsymbol{W_{fx}}\boldsymbol{x_t} + \boldsymbol{b_f})$$

Input gate decides which values we'll update

$$\boldsymbol{i_t} = \sigma(\boldsymbol{W_{ih}}\boldsymbol{h_{t-1}} + \boldsymbol{W_{ix}}\boldsymbol{x_t} + \boldsymbol{b_i})$$

New candidate cell state is to decide what new information we're going to store in the cell state.

$$\widetilde{\boldsymbol{c}}_{\boldsymbol{t}} = tanh(\boldsymbol{W_{ch}}\boldsymbol{h_{t-1}} + \boldsymbol{W_{cx}}\boldsymbol{x_t} + \boldsymbol{b_c})$$

New cell state:

$$\boldsymbol{c_t} = \boldsymbol{f_t} * \boldsymbol{c_{t-1}} + \boldsymbol{i_t} * \widetilde{\boldsymbol{c}}_{\boldsymbol{t}}$$

Output Gate decides what parts of the cell state we're going to output.

$$\boldsymbol{o_t} = \sigma(\boldsymbol{W_{oh}}\boldsymbol{h_{t-1}} + \boldsymbol{W_{ox}}\boldsymbol{x_t} + \boldsymbol{b_o})$$

Output:

$$\boldsymbol{h_t} = \boldsymbol{o_t} * tanh(\boldsymbol{c_t})$$

13.5 Gated Recurrent Units

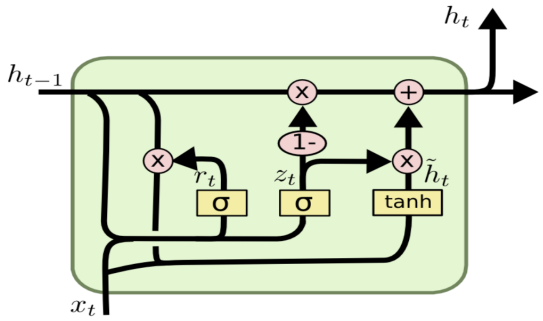


Figure 13.3: Gated Recurrent Units Source

A slightly more dramatic variation on the LSTM is the Gated Recurrent Unit Cho et al. [2014]. It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models. Below is how GRU works

The update gate decides what information from the past would be passed to the next cell state.

$$\boldsymbol{z_t} = \sigma(\boldsymbol{W_{zh}}\boldsymbol{h_{t-1}} + \boldsymbol{W_{zx}}\boldsymbol{x_t} + \boldsymbol{b_z})$$

The reset gate determine what information would be discarded from previous cell states.

$$\boldsymbol{r_t} = \sigma(\boldsymbol{W_{rh}}\boldsymbol{h_{t-1}} + \boldsymbol{W_{rx}}\boldsymbol{x_t} + \boldsymbol{b_r})$$

Then the current memory content update the stories the latest important information by using the reset gate.

$$\widetilde{\boldsymbol{h}}_{\boldsymbol{t}} = \tanh(\boldsymbol{W}_{\boldsymbol{hr}}(\boldsymbol{r}_{\boldsymbol{t}} * \boldsymbol{h}_{\boldsymbol{t-1}}) + \boldsymbol{W}_{\boldsymbol{hx}}\boldsymbol{x}_{\boldsymbol{t}} + \boldsymbol{b}_{\boldsymbol{h}})$$

Finally, the final cell state updates the information gained from the current unit and passed it to the next cell.

$$\boldsymbol{h}_{\boldsymbol{t}} = (1 - \boldsymbol{z}_{\boldsymbol{t}}) * \boldsymbol{h}_{\boldsymbol{t-1}} + \boldsymbol{z}_{\boldsymbol{t}} * \widetilde{\boldsymbol{h}}_{\boldsymbol{t}}$$

Chapter 14

Recurrent Neural Networks

14.1 High-Level Overview of Recurrent Neural Networks

As Convolutional Neural Networks naturally lend themselves to images, RNNs (including LSTMs and Transformers) lend themselves similarly to sequential data (which could additionally include each step being images). These might take the form of time series, speech recognition, learning grammars (language translators, name generators, etcetera) or musical rhythms - among others. More precisely one can define them to be directed graphs mapping to a temporal sequence.

Visual captioning Vinyals et al. [2016] is another interesting application, where they combine RNNs and CNNs in order to attempt to generate a text description of a scene 21.1

14.1.1 Sequence to vector

Explanation: input: a sequence of words; output: a scalar value or a vector (see Figure 14.2).

Example: learning to execute (Zaremba & Sutskever, 2015)
Neural network is trained to interpret several lines of python codes and report the result of the running code (see Figure 14.3).

14.1.2 Sequence to vector to sequence

Explanation: a sequence of words → hidden representations(vector) → output sequence (see Figure 14.4).

Example: phrase representation clustering (Cho et al., 2014)
Semantically close phrases are close to each other on the representation map (Figure 14.5)
One interesting characteristic is one could perform arithmetic calculation of the semantics in hidden variable space. For example, we have

$$h('king') - h('men') + h('women') = h('queen')$$

Figure 14.1: Vinyals et al. (2016) Show & Tell: Lessons Learned from the 2015 MSCOCO Image Captioning Challenge

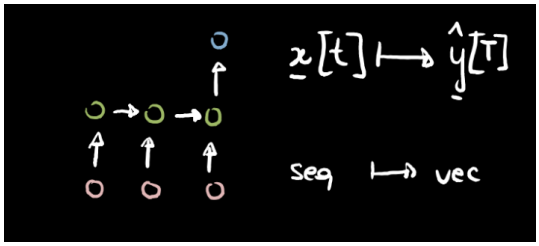


Figure 14.2: A hand-drawn diagram showing the sequence to vector rationale in the neural networks. Red, green, blue circles are input,hidden,output nodes, respectively. A sequence of inputs are given to the network and there is only one output in the end.

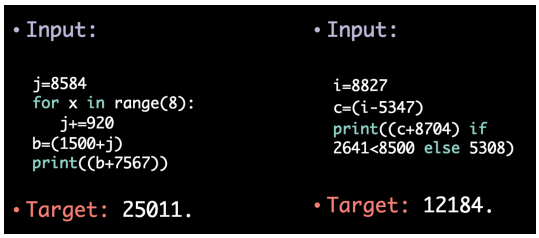


Figure 14.3: An example of sequence to vector rationale. A paragraph of Python code is given to the network as an input and the execution output of the Python code is supposed to be the output of the network.

where h(X) is the hidden state of word 'X'.

14.1.3 Sequence to sequence

Explanation: a sequence of words → another sequence of words (see Figure 14.6).

Example: auto completion: Once a character is typed, there will be a new sequence of suggestions of auto completion.

Example: RNN writer(Sloan(2016))
Neural network is trained on sci-fi novels and could be used to auto complete the sci-fi novel.
Visit github.com/robinsloan/rnn-writer for more details.

14.2 RNN training

14.2.1 BPTT: backpropagation through time

In order to train a RNN, we are supposed to use backpropagation through time(BPTT) (see Figure 14.7). In order to calculate the hidden state value h[t], we have

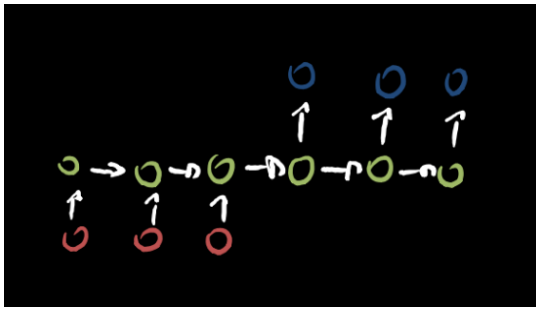


Figure 14.4: A hand-drawn diagram showing the sequence to vector to sequence rationale in the neural networks. Red, green, blue circles are input,hidden,output nodes, respectively. A sequence of inputs are given to the network and used to compute hidden state. Hidden state variable are then used to generate a sequence of outputs.

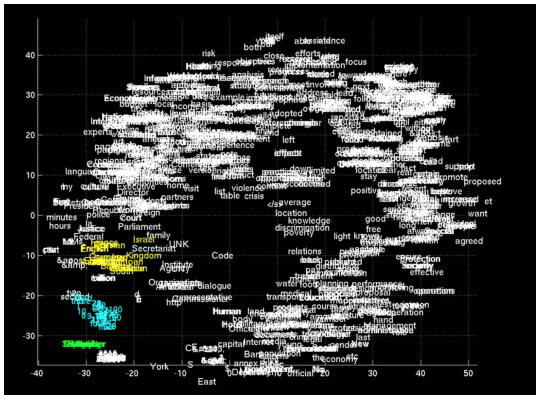


Figure 14.5: An example of sequence to vector to sequence rationale. Words are clustered according to their semantics

$$h[t] = g(W_{hx}x[t] + W_{hh}h[t-1] + b_h) \tag{14.1}$$

To simplify the equation, we define W_h as

$$W_h = \begin{bmatrix} W_{hx} & W_{hh} \end{bmatrix} \tag{14.2}$$

Thus, we could rewrite equation 1 as

$$h[t] = g(W_h \begin{bmatrix} x[t] \\ h[t-1] \end{bmatrix} + b_h) \tag{14.3}$$

$\hat{y}[t]$ could be calculated as shown in figure.14.8 and then we could use normal backpropagation algorithm except that we sum up the gradients at each time step.

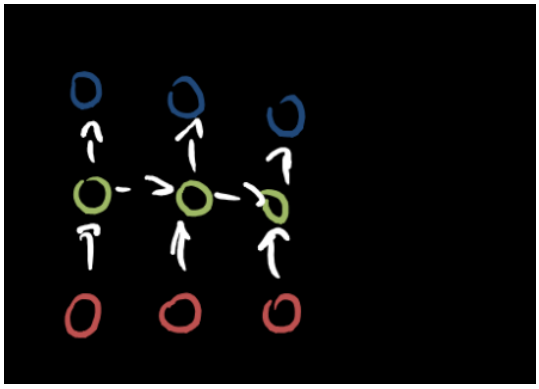


Figure 14.6: A hand-drawn diagram showing the sequence to sequence rationale in the neural networks. Red, green, blue circles are input, hidden, output nodes, respectively. A sequence of inputs are given to the network and used to compute hidden state. Hidden state variable are used to generate a sequence of outputs in real time.

14.3 Batch-ification

When dealing with text sample with large size, we could slice the text into different batches. For example, the batch-ification process for the following sequence of words should exhibit:

$$\begin{bmatrix} a & g & m & s \\ b & h & n & t \\ c & i & o & u \\ d & j & p & v \\ e & k & q & w \\ f & l & r & x \end{bmatrix}$$

In this sequence of words, the batch size is 4. We then determine the input and output for our recurrent neural network, namely:

$$X[1:T] = \begin{bmatrix} a & g & m & s \\ b & h & n & t \\ c & i & o & u \end{bmatrix}$$

$$Y[1:T] = \begin{bmatrix} b & h & n & t \\ c & i & o & u \\ d & j & p & v \end{bmatrix}$$

Notice here $T = 3$ as we proceed three sequence of words. The next step is to perform back-propagation through time on the sequences of words, which is to say we calculate gradient descent horizontally and vertically. Every time we extract one word from the input sequence and we are aiming to predict the corresponding output in the output sequence. For example, when we select $X[1] = [a, g, m, s]$ as input, we train through the RNN to predict $Y[1] = [b, h, n, t]$. At the same time, we send the

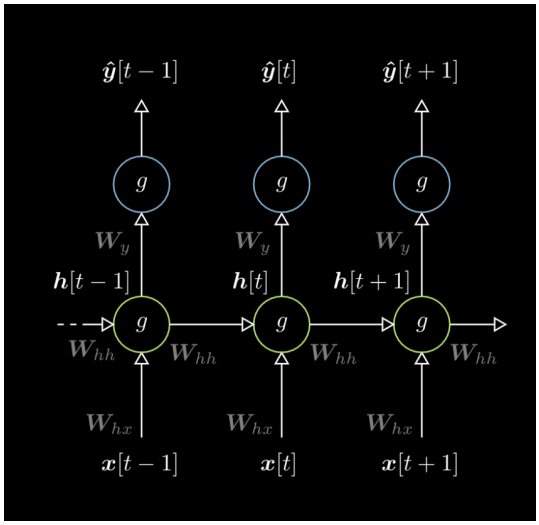


Figure 14.7: A diagram showing how the values are propagated in a RNN. At each time point t, y is calculated using h from the previous time step and current x. g() is the transfer function, ReLu for example. h[t] is the value of hidden layer at time t. The initial state of h[t] is 0.

hidden representation h[1] forward to perform the RNN given $X[2]$ predicting $Y[2]$. After sending the final hidden representation $h[T-1]$ for the final set of $X[T]$ and $Y[T]$, we cut the gradient and conclude the temporal bulk of this batch.

The reason behind concluding the temporal bulk after we sending the last hidden representation of the RNN is that we need to stop the gradient descent eventually. Thus, the gradient descent from the following sequences of words is zero.(The PyTorch Auto-grad will handle this critical point.) Similarly, we also don't have the gradient coming from the past sequences before this batch.

14.4 Vanishing of the Gradient and GRU

The limitation of a recurrent Neural Network is the diminishing of the gradient back in time. A very strong gradient would be weakening back through time because of the way we initialize the weight-by selecting very small eigenvalues across zero. A powerful technique that preventing the gradient from vanishing back through time is by employing Gated Recurrent Unit (GRU).

GRU uses two gates-update gates and reset gates. A update gate helps the model to determine how information/gradient from the previous time steps need to pass along to the future. A reset gate, on the other hand, determine how much past information/gradient to forget.

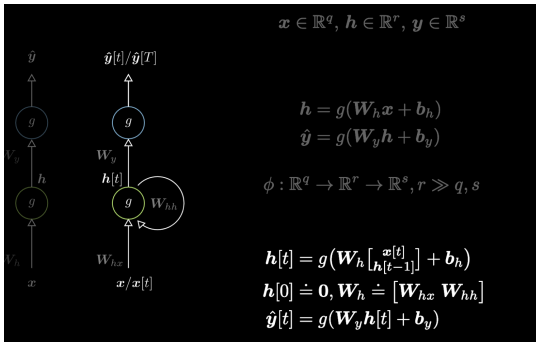


Figure 14.8: $\hat{y}[t]$ shares the same formula as traditional feedforward network. All we have to do is to use the chain rule and backpropagate the error to previous time step.

14.5 Long Short Term Memory

Long Short Term Memory, or typically referred to as LSTMs - these are second order RNNs, tend to imbue a sense of locality (spatial, temporal) to the network models.

The forward propagation of the RNN are represented by:

$$
\begin{aligned}
h[t] &= g(W_h \begin{bmatrix} x[t] \\ h[t-1] \end{bmatrix}) + b_h \\
\bar{y}[t] &= g(W_y h[t] + b_y)
\end{aligned}
$$

The decomposition of a Gated Recurrent Unit (GRU) will be visualized as:

$$
\begin{aligned}
i[t] &= \sigma(W_i \begin{bmatrix} x[t] \\ h[t-1] \end{bmatrix}) + b_i \\
f[t] &= \sigma(W_f \begin{bmatrix} x[t] \\ h[t-1] \end{bmatrix}) + b_f \\
o[t] &= \sigma(W_o \begin{bmatrix} x[t] \\ h[t-1] \end{bmatrix}) + b_o \\
\bar{c}[t] &= \tanh(W_c \begin{bmatrix} x[t] \\ h[t-1] \end{bmatrix}) + b_c \\
c[t] &= f[t] \odot c[t-1] + i[t] \odot \bar{c}[t] \\
h[t] &= o[t] \odot \tanh(c[t])
\end{aligned}
$$

To control the output, take saturated (squashing / squeezing inputs to a range) sigmoid as an example which we could make the switch simply as 1 or 0, we could determine the hidden representation to be zero by setting the sigmoid function to be null; On the other hand, we could output accordingly by setting the sigmoid function to be 1.

To control the memory, we could manipulate the input network and the "don't forget" gate. Simply setting them both to zero, we could reset the memory. To keep the memory, we set the input unit to be zero and memory to be one using the sigmoid function to keep the memory without adding anything from the current input. Finally, we could write to the memory by setting the saturated sigmoid function both to be one to add input to the memory.

14.6 Transformers? Pay Attention!

A new architecture has found popularity in many applications typically dominated by RNNs and LSTMs, Transformers - introduced by Vaswani et al. [2017]. Transformers essentially add attention RNNs and LSTMs suffer issues with the memory bandwidth bound computations in training, where given the orthogonality of traditional convolutional networks to the GPU hardware, CNNs map perfectly to GPUs. Contrarily these hierachical (sometimes) sequential representations don't typically map quite so easily, and hence, performance suffers as a result.

Chapter 15

Optimization in Deep Learning

Optimization is one of the important aspects of deep learning. In this chapter, we talk about the basics of parameter updating and the step size. Choosing a right step size can greatly reduce the training time.

15.1 Gradient Descent in 1-Dimension

Gradient descent is one of the most popular optimization algorithms used in Deep Learning. We initialize weights randomly and update in the direction in which the value of the loss function E descends the steepest in order to find the minimizer w. This direction can be found by taking the negative of the gradient. This process can be represented as:

$$w = w - \eta \frac{\partial E}{\partial w}$$

where

$$\begin{aligned} w &= \text{weight} \\ \eta &= \text{learning rate} \\ \frac{\partial E}{\partial W} &= \text{gradient of loss function with respect to the weight} \end{aligned}$$

The time until convergence by the algorithm is related to (but not limited to) the learning rate, as given by the formula. For instance, we consider a quadratic loss function to optimize. This function is convex and there exists a unique minimizer. The figure below represents the different scenarios for different step sizes.

- For $\eta < \eta_{\text{opt}}$, the update takes small steps and longer time to the minimum.
- For $\eta = \eta_{\text{opt}}$, we have a convergence to the minimizer in one step.
- For $\eta > \eta_{\text{opt}}$, it bounces off the wall but still converges.
- For $\eta > 2\eta_{\text{opt}}$, the update overshoots and we do not have convergence.

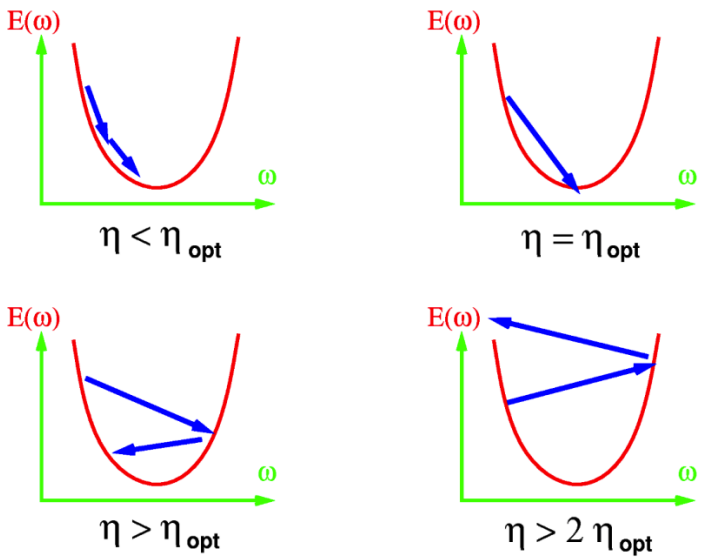


Figure 15.1: Various scenarios of tuning the learning rate.

The question arises: "What's the optimal value for the learning rate η?" The optimal step size is given by computing the second derivative of the loss function.

$$\eta_{opt} = (\frac{\partial^2 E}{\partial w^2})^{-1}$$

Therefore the weight update rule in this case is given by:

$$w_{t+1} = w_t - (\frac{\partial^2 E}{\partial w_t^2})^{-1} \frac{\partial E}{\partial w_t}$$

This is given by Newton's algorithm for optimization[1]. Above example is for 1-Dimensional case, but it can also be generalized to multiple dimensions.

15.2 Gradient Descent in n-Dimension

In this section, we observe the gradient descent in multiple dimensions. Let m be the dimension of input space and let us consider a single linear unit with m-dimensional inputs. The quadratic loss function is given as follows:

$$E(\boldsymbol{W}) = \frac{1}{P} \sum_{p=1}^{P} (Y_p - \boldsymbol{W}^\top \boldsymbol{X}_p)^2$$

where p refers to the index of the training data. We want to compute the gradient and the second derivative.

[1] https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization

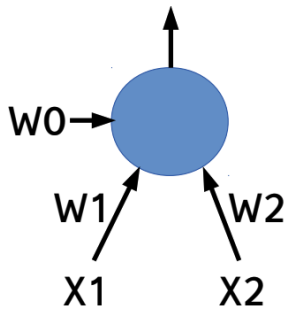


Figure 15.2: A single linear unit with 2-dimensional input.

$$
\begin{aligned}
E(\boldsymbol{W}) &= \frac{1}{P}\sum_{p=1}^{P}(Y_p - \boldsymbol{W}^\top \boldsymbol{X}_p)(Y_p - \boldsymbol{W}^\top \boldsymbol{X}_p) \\
&= \frac{1}{P}\sum_{p=1}^{P} Y_p^2 - 2Y_p \boldsymbol{W}^\top \boldsymbol{X}_p + \boldsymbol{W}^\top \boldsymbol{X}_p \boldsymbol{X}_p^\top \boldsymbol{W}
\end{aligned}
$$

Taking the first and second derivative of $E(\boldsymbol{W})$ with respect to $\boldsymbol{W}$ gives us:

$$
\begin{aligned}
\frac{\partial E}{\partial \boldsymbol{W}} &= -\frac{2}{P}\sum_{p}(Y_p - \boldsymbol{W}^\top \boldsymbol{X}_p)\boldsymbol{X}_p^\top \\
\frac{\partial^2 E}{\partial \boldsymbol{W}^2} &= \frac{2}{P}\sum_{p}\boldsymbol{X}_p \boldsymbol{X}_p^\top
\end{aligned}
$$

The second derivative turns out to be a covariance matrix of the input vectors, shaped as a symmetric matrix with $m \times m$ dimension. This matrix is also called as the Hessian Matrix, represented by $\boldsymbol{H}$. Therefore the update rule is now given by:

$$
\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \boldsymbol{H}^{-1}\frac{\partial E}{\partial \boldsymbol{W}_t}
$$

The main takeaways are as follows. With the right learning rate, the gradient doesn't paddle left or right, and this also leads to the convergence to the global minimizer. Another thing to note is that when we tune the learning rate, it affects the speed of training but it is usually hard to find the optimal learning rate beforehand.

15.3 Cost Function Appearance and Eigen-Decomposition

From the previous section we observed the quadratic cost function.

$$E(\boldsymbol{W}) = \frac{1}{P}\sum_{p=1}^{P} Y_p^2 - 2Y_p\boldsymbol{W}^\top\boldsymbol{X}_p + \boldsymbol{W}^\top\boldsymbol{X}_p\boldsymbol{X}_p^\top\boldsymbol{W}$$

$$\nabla E(\boldsymbol{W}) = \frac{1}{P}\sum_{p=1}^{P} -2Y_p\boldsymbol{X}_p + 2\boldsymbol{X}_p\boldsymbol{X}_p^\top\boldsymbol{W}$$

$$H = \nabla^2 E(\boldsymbol{W}) = \frac{2}{P}\sum_{p=1}^{P} \boldsymbol{X}_p\boldsymbol{X}_p^\top$$

The Taylor expansion of E at the minimizer $\boldsymbol{W^*}$ is exact (no remainder term) and we get

$$E(\boldsymbol{W}) = E(\boldsymbol{W^*}) + (\boldsymbol{W} - \boldsymbol{W}^*)^\top\boldsymbol{H}(\boldsymbol{W} - \boldsymbol{W}^*)$$

The function E is a quadratic function with respect to each input, and the Hessian matrix determines the curvature and orientation of each dimension. We study the diagonalization (eigen-decomposition) of H:

$$\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$$

where $\boldsymbol{Q}$ is an orthogonal matrix (rotation into eigen-spaces) and $\boldsymbol{\Lambda}$ is a diagonal matrix (scaling in eigen-spaces). The gradient is orthogonal to the level sets (lines of equal cost) and the gradient is larger in a dimension where the curvature is larger.

In practice, computing the Hessian is intractable. Even if it were, it might not be positive definite. Thus applying Newton's method to Deep learning might be hopeless, but there are other smart tricks: transforming the data and network in such a way that $\boldsymbol{H}$ has good properties.

15.4 Examples with Input Characteristics Combinations

15.4.1 Case 1: Best Case

Given $\mu_{x_1,x_2} = 0$, $\sigma^2_{x_1,x_2} = 1$, and x_1 and x_2 are uncorrelated, this would result into a Hessian matrix being the identity matrix.

$$\boldsymbol{H} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

This is the best case as it means learning rate can be the same for both inputs x_1 and x_2, but what does it mean if the inputs have different characteristics?

15.4.2 Case 2 : inputs have varying variances/standard deviations

Given $\mu_{x_1,x_2} = 0$, $\sigma_{x_1} = 1$, $\sigma_{x_1} = 2$, and x_1 and x_2 are uncorrelated, this would result into a Hessian with the following form:

$$\boldsymbol{H} = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}$$

This is problematic! Because the optimum learning rate for x_1 is $\frac{1}{4}$ of that of x_2, the maximum learning rate is limited to 4. Thus x_1 learns 4 times slower. The moral of the story is to make sure the input variable's variance is more or less the same or very close, commensurate. Things Professor LeCun mention in passing:

- A lot of pre-processing is done by data loaders to ensure that variances normalize to one for standard datasets.
- For images, don't normalize all pixels individually. Normalize every channel by a single constant.

15.4.3 Case 3: inputs resulting to different means

Given $\mu_{x_1,x_2} = 0$, $\sigma^2_{x_1,x_2} = 1$, and x_1 and x_2 are uncorrelated but x_1 and x_2 has 100 added to them, this would result into a Hessian with the following form:

$$\boldsymbol{H} = \begin{bmatrix} 10001 & 10000 \\ 10000 & 10001 \end{bmatrix}$$

This is terrible! This results to eigenvalues of 10,000 and 1 which result to a curvature that is 10,000 times larger in one direction compared to another.

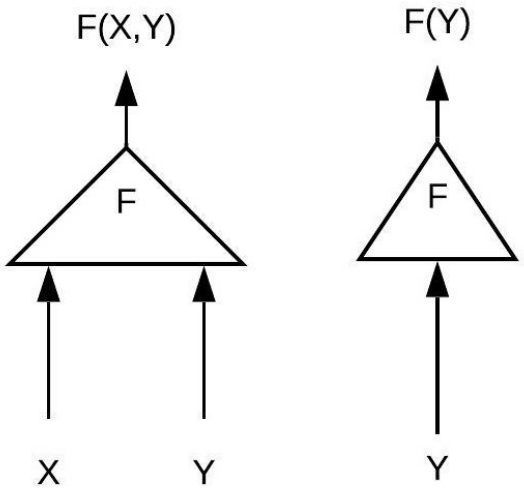
Chapter 16

Energy-Based Models

16.1 Introduction

The Energy-Based Models (EBM) approach provides a general framework for many machine learning algorithms. They are designed to deal with situations where the machine learning system does not just compute an output as in a feed-forward process, but computes an output as the result of some optimization.

There are two forms of EBM, one for supervised learning (conditional) and one for unsupervised learning (unconditional). The conditional models take x and y as input, while the unconditional models take only y as input. Their outputs are scalars, $F(x, y)$ or $F(y)$.

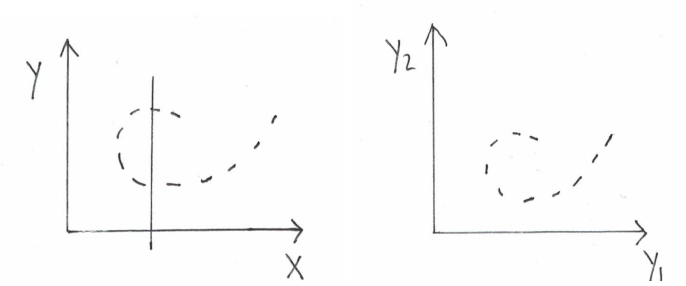


(a) Conditional (b) Unconditional

Figure 16.1: Two forms of Energy-Based Models

The output value indicates the "compatibility" of the input. In the conditional case, assume we have a properly trained EBM, and we show it an image of a table and the label for table, we get a small value as output, indicating these two are compatible. If we use an image of a table and the label for car, then the output energy should be higher. In the unconditional case, there is only one input, and the model tells us if it "looks good". For example, if this model is trained on natural images (ImageNet), and we show it an image like those in ImageNet, we get a low energy. But if we show it noise or something not natural, we get a high energy.

The model can be viewed as an energy function F. Suppose x and y are both scalars, and the data points are two-dimensional like in 16.2a and 16.2b. In the conditional case, We want a function that takes low values on observed points and high values elsewhere. The problem is that for the value of x at the bar, there are two possible y. If we build a model, for example a neural network, and compute y, we can only get one value for y. How do we represent the fact that we have two y? One trick is that we go through an implicit function. When we want to parameterized a circle, we use $x^2 + y^2 = r^2$. We can design an energy function that looks like $(x^2 + y^2 - r^2)^2$, which takes value 0 on the circle and larger values inside or outside the circle. This is one problem EBM attempts to solve. In the unconditional case, instead of x and y, we have two y. Instead of predicting y from x, we might have to predict y_1 from y_2 or y_2 from y_1 or just how they are compatible with each other.



(a) Conditional (b) Unconditional

Figure 16.2: Usage of EBM

For the above problems, we want some energy function that looks like 16.3. The darker area represents the valley. When we move away from the valley, our energy goes up. For predicting y from x, We can just find where our y has the lowest energy.

16.1.1 Relation with Probabilistic Models

If we want to turn an EBM into a probabilistic model, like $p(y|x)$. We can parameterized it with:

$$P(y|x) = \frac{e^{-\beta F(x,y)}}{\int_y e^{-\beta F(x,y)}}$$

Now we are turning an energy which we have not specified positive or negative into positive. Here the integral is over the entire space of y, so this is a normalized distribution over y. This is called the (Gibbs) Boltzmann distribution, which comes from statistical physics. This is a way of transforming an energy function into a distribution. But the normalization term is not always

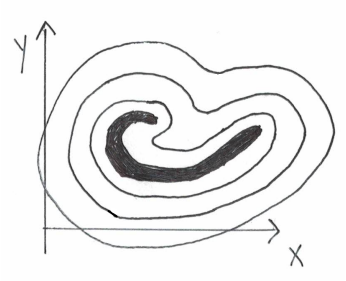


Figure 16.3: Energy Function

tractable. y might be in high-dimension space. The energy function does not have a easy-to-compute integral, and so the normalization term is not computable. We have to use tricks like variational methods or Markov chain Monte Carlo. But this would make it complicated.

16.1.2 Energy-Based Inference

In the conditional case, the inference process is to find a y which minimizes this F function. like

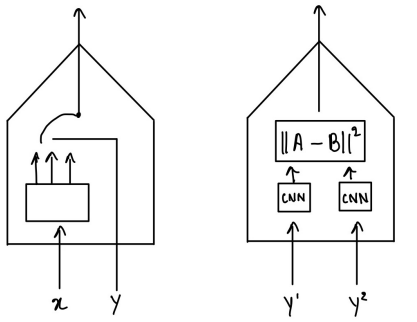
$$y^* = \arg\min_y F_w(x, y)$$

Note that there can be multiple minima.

When designing the F function, its range can be lower bounded, like the square function, so we know 0 is the floor, and can make it larger for other things.

Another kind of inference would be returning a distribution over y instead of a single y. But the machine has to be trained a particular way.

The F function is not minimized during training, but instead during inference. For training, we shall minimize another loss function. We are going to parameterized F with w and train it.



(a) Conditional (b) Unconditional

Figure 16.4: Two examples of Energy-Based Models

In 16.4a, the EBM takes an image, and run through a CNN which produces a bunch of scores for some categories. The y is a discrete variable, which acts like a switch and chooses which output of

the CNN is propagated to the final output. Suppose we have three categories: table, car, airplane. When we show it an image of car, and put car in y, it will select the output of car from the CNN, and return a score. A high score is bad, and a small score is good. If we put the label table, car and airplane in y, we select each of those, and get their scores. If we want to know which one it is, we find the one which gives the lowest energy, and this will be the answer.

In 16.4b, we feed two vectors y_1 and y_2, and ask the system if they are compatible. We run them both through CNN, and compute the distance between the two outputs. This can be used in a face detector, where the model gives low energy for faces of the same person and high energy for faces of different person.

16.2 Energy-Based Training

The training process should return a function like this: When we have training samples like (x_i, y_i), we want $F(x_i, y_i)$ to be small. At the same time, we want $F(x_i, y_j)$ to be large, where $i \neq y$. For the unconditional case, we want $F(y_i)$ to be small.

$$Conditional \begin{cases} F_w(x^i, y^i), & \text{small.} \\ F_w(x^i, y^j), & i \neq j, \text{larger.} \end{cases}$$

$$Unconditional \begin{cases} F_w(y^i), & \text{small.} \\ F_w(y) & y \neq y^i, \text{ larger.} \end{cases}$$

We need an objective function which push down the energy of training samples, and another term that pushes up the energy of everything else (outside the training samples).

Suppose we want an energy function that looks like 16.3. The training samples will be in the valley region. After getting a training sample, we can tweak the parameters of the energy function by gradient descent, so that the output energy goes down.

How do we push up the energy outside? We can generate samples outside and push them up. The space outside can be large, but this is what probabilistic approaches do.

Say we believe in maximum likelihood. Suppose we parameterized $p(y|x)$ with a Boltzmann distribution.

$$P_w(y|x) = \frac{e^{-\beta F_w(x,y)}}{\int_y e^{-\beta_w F(x,y)}}$$

We want to maximize the probability over the training samples

$$\prod_i P(y^i|x^i)$$

That is, we want to minimize the negative log of it.

$$L(w) = -\sum_i \log P(x^i|y^i)$$

That is our objective function. We plug in the distribution and rename this function, we get

$$
\begin{aligned}
L^{'}(w) &= \sum_i -\log\left[\frac{e^{-\beta F_w(x,y)}}{\int_y e^{-\beta_w F(x,y)}}\right] \\
&= \sum_i [-\log e^{-\beta F(x,y)} + \log \int_y e^{-\beta F_w(x,y)}]
\end{aligned}
$$

$$
\begin{aligned}
L(w) &= \frac{1}{\beta} L^{'}(w) \\
&= \sum_i [F_w(x^i, y^i) + \frac{1}{\beta} \log \int_y e^{-\beta F_w(x,y)}]
\end{aligned}
$$

When we try to minimize this, the first term tries to make the energy of the correct y small, while the second term will make the energy of all y high.

If we compute the gradient with respect to w

$$
\frac{\partial L(w)}{\partial w} = \sum_i [\frac{\partial F_w(x^i, y^i)}{\partial w} - \int_y P_w(y|x^i) \frac{\partial F_w(x^i, y)}{\partial w}]
$$

where $P_w(y|X^i)$ is obtained through the Gibbs distribution.

There is a negative sign before the second term, which means it will increase the energy of all the y to some extent. More generally, what the gradient indicates is that, we are pushing down on the energy of the correct answer in the first term, and pushing up the energy of every answer, including the correct one, in the second term. And the force we push up a particular y is proportional to the probability our model gives to that y. Note that the first term pushes down the energy of correct answers with a larger force.

If you have the energy function and can calculate the normalized term, then this is the way for maximizing the likelihood of the data, using the negative log-likelihood function.

If we take the EBM for 16.4a, and we assume the probability of one y given by the model is according to the Gibbs distribution. Suppose we try to push down the energy of the correct answers and push up the energy of the incorrect answers. Then this objective function becomes negative log-softmax, and this model is equivalent to a softmax on top of a neural network. We can also use other loss functions with similar results.

As it is difficult to parameterize F, we may use Monte Carlo methods to approximate the integral, for example using $\sum_k \frac{\partial F_w(x_i, y^k)}{\partial w}$. Another form of Monte Carlo is called Markov chain Monte Carlo (MCMC), where we have a sample y, and we modify the sample each time in such a way that every modification gives a sample that is more likely under the distribution. Another category for approximation is called variational methods, where we assume P to have simpler form (e.g. Gaussian).

Basically, any object function that pushes down the energy of correct answers and pushes up the energy of incorrect answers will work. It may not be related to probabilities.

Chapter 17

Latent Variable Models

17.1 Latent Variable Models: Face Detection

In this section, we take a look at an example of latent variable models in object detection. Consider the following figure. Say we want to train the system for face detection. In this model,
X: observed variables (input variables)
Y: observed on training set (output variables)
Z: never observed (latent variable)

We input an image where which may or may not contain a face. The location of the face is also unknown to us so, Z can be considered a latent variable which represents the location of the face.

For the sake of the example, let's imagine that this system is composed of a convolutional neural network(CNN) and an energy based model(EBM) on top of it. Using a switch, for values smaller than the threshold T, we decide there is a face and there isn't for others. Because we intend to minimize energy, low values are good and high values are bad for the score. The CNN run overs all the candidate locations of the face and gives a score based on that and the EBM minimizes the energy with respect to Z and Z is also computed in this process which is the model's best guess about the location of the face.

Our goal is to find the combination of Z and Y that minimises the energy. T This is a good approach for unsupervised learning where we do not know where the face is located in the image. System is only stable when it picks the face and the learning can happen through gradient descent. Marginalization is better because you would not need a switch anymore and it would be a continuous module that you can back propagate through, so learning happens in all scenarios.

Generally there are two methods involved which are minimizing energy with respect to Z and marginalize energy with respect to Z. The integral for marginalization often requires a lot of approximations.

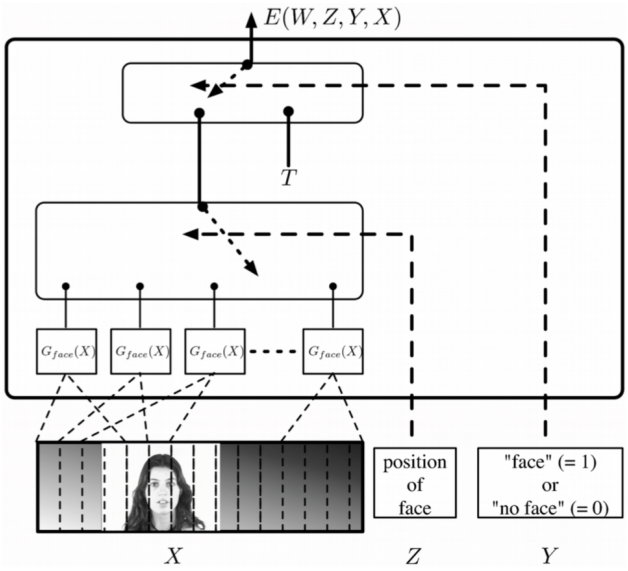


Figure 17.1: Training the system for face detection and localisation

17.2 Latent Variable Models: Integrated training with sequence alignment

Let's look at another example of latent variable models. Say we want to detect a few keywords - Alexa, Hey Google and so on.

These systems consume minimum energy because they run all the time. We can have templates but these templates don't have definite length because the speed of speaking same words is different for different people. Elastic template matching solves this problem and it was previous used in speech recognition systems. You have to minimize the energy for multiple templates and then, you output the template with lowest energy.

You can do it using dynamic programming by matching up all the features and template features and creating a matrix of distances. After that, you have to traverse through it and choose the path with lowest cost.

We create a template for each keyword and follow this process:

1. Input the sequence
2. Extract trainable features to get a sequence of feature vectors
3. Then we do elastic matching with the input and the template.
4. We do this for every template and then output the template that has the lowest energy.

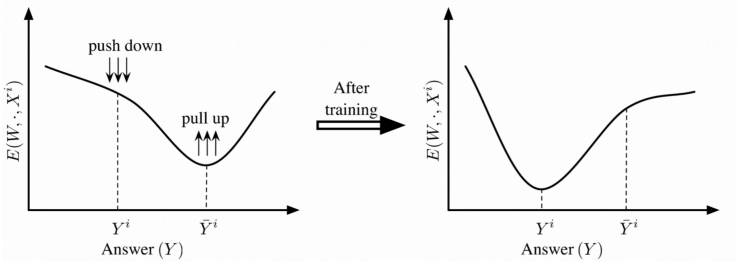


Figure 17.2: Loss function before training (left) and after training (right)

17.3 What can latent variables represent?

1. Face recognition: Gender, orientation of the face.
2. Object recognition: Pose parameters of the object, lighting conditions.
3. Speech Recognition: Segmentation of the sentence into phonemes or phones.
4. Handwriting Recognition: segmentation of line into characters.
5. Scene Parsing: Segmentation of image into components.
6. Parts of speech tagging: Segmentation of sentence into syntactic units.

17.4 Loss Function

We design a loss function in such a way that the model gives low energy to the inputs that we want and high energy(i.e. cost) to the inputs that we don't want. We essentially aim to push down on the energy of the correct answer and pull up on the energies of the incorrect answers.

Note: A simple energy loss function faces a "collapse" problem. This is something to pay attention to at the time of training.

17.4.1 A better loss function: Generalised Margin Losses

As long as the energy of the incorrect answers is very large as compared to the right answer, the collapse problem won't be there. More specifically, if the energy of the "best wrong answer" is higher than the energy of the correct answer, then the model should work. For e.g. Square Square loss (Figure 17.3)

17.5 Approach for EBM models

These are the steps:
1. Design an architecture choosing a particular form of energy function E(W,y,X)
2. Pick an inference algorithm for Y
3. Pick a loss function: In such a way that minimising it with respect to W over a training set will make the inference algorithm find the correct Y for X.
4. Pick an optimisation method.

The simplest loss function is the energy itself that the model assigns.

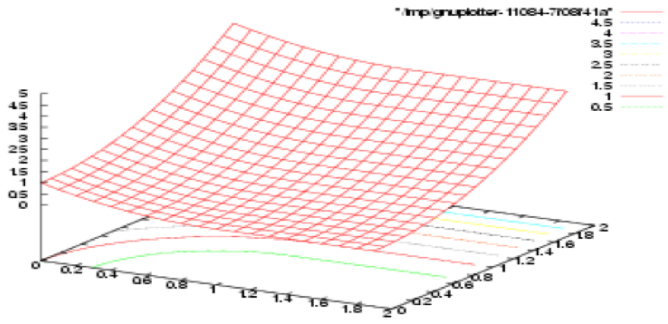


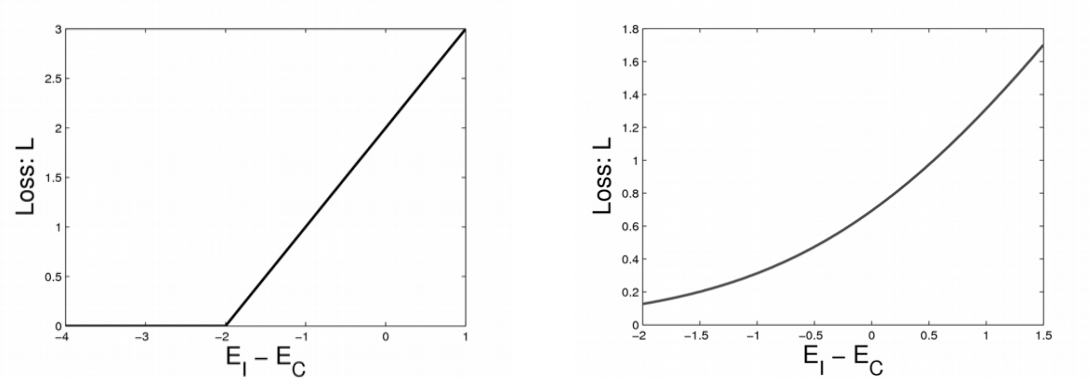
Figure 17.3: Square-Square Loss

Consider the example where energy is defined as the L2 norm of difference between outputs of two neural networks. One is fed with X and other with Y. In this scenario, the simplest way for the system to minimize energy is to assign zero weights to both networks and a similar bias term and get energy zero. This is called a collapse.

When calibrated energies at the output are not useful and you are only concerned with making a decision, then you only care about the energy of the correct answer being lower than the rest, the actual values themselves don't matter.

Perceptron loss is a special condition of negative log likelihood.

If an energy function is a linear function of parameters, then we don't have a collapse problem. But if it is non-linear, there are chances of a collapse. But even then, as long as the energy of the most offensive incorrect answer is larger than the correct answer, we won't have a collapse. Hinge loss and log loss are examples of such a loss function.



(a) Hinge Loss (b) Log Loss(Soft-Hinge Loss)

Figure 17.4: Examples of Generalized Marginal Losses

The table below shows various good and bad loss functions.

Loss (equation #)	**Formula**	**Margin**
energy loss	$E(W, Y^i, X^i)$	none
perceptron	$E(W, Y^i, X^i) - \min_{Y \in \mathcal{Y}} E(W, Y, X^i)$	0
hinge	$\max\left(0, m + E(W, Y^i, X^i) - E(W, \bar{Y}^i, X^i)\right)$	m
log	$\log\left(1 + e^{E(W,Y^i,X^i) - E(W,\bar{Y}^i,X^i)}\right)$	> 0
LVQ2	$\min\left(M, \max(0, E(W, Y^i, X^i) - E(W, \bar{Y}^i, X^i)\right)$	0
MCE	$\left(1 + e^{-\left(E(W,Y^i,X^i) - E(W,\bar{Y}^i,X^i)\right)}\right)^{-1}$	> 0
square-square	$E(W, Y^i, X^i)^2 - \left(\max(0, m - E(W, \bar{Y}^i, X^i))\right)^2$	m
square-exp	$E(W, Y^i, X^i)^2 + \beta e^{-E(W,\bar{Y}^i,X^i)}$	> 0
NLL/MMI	$E(W, Y^i, X^i) + \frac{1}{\beta} \log \int_{y \in \mathcal{Y}} e^{-\beta E(W,y,X^i)}$	> 0
MEE	$1 - e^{-\beta E(W,Y^i,X^i)} / \int_{y \in \mathcal{Y}} e^{-\beta E(W,y,X^i)}$	> 0

Figure 17.5: Good and bad Loss functions

Chapter 18

Regularisation and Overfitting

18.1 Definitions of regularisation

(1) Regularisation adds prior knowledge to a model i.e. a prior distribution is specified for the parameters

(2) Restriction of the set of possible learnable functions

(3) "Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error" – Ian Goodfellow (Goodfellow et al. [2016])

Definition (3) is commonly used in the context of machine learning (and deep learning). Intuitions for regularisation and overfitting can be seen in Figure 18.1.

18.2 Techniques

The following are the regularisation techniques used in neural networks. These techniques also reduce overfitting.

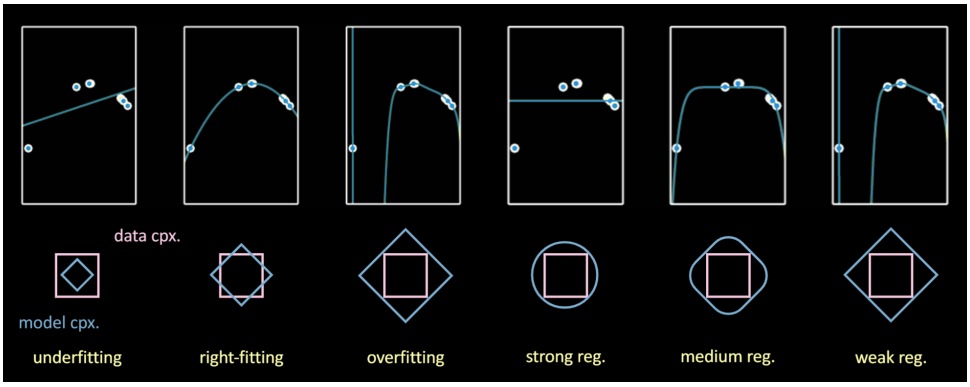


Figure 18.1: Regularisation and overfitting

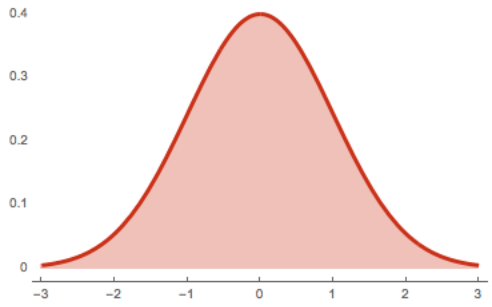


Figure 18.2: Normal distribution with zero mean

18.2.1 Xavier initialization

Random initialization of weights of a neural network before start of training can potentially lead to issues of vanishing gradients or exploding gradients. This in turn can cause the network to learn specific patterns instead of generalizing. This can be avoided using Xavier initialization.

In Xavier initialization, the weights of the network are initialized based on some distribution. The distribution used is typically Gaussian (as in Figure 18.2) or uniform. The original paper Glorot and Bengio [2010] suggested to initialize the weights W of a layer from a distribution with zero mean and variance given by

$$Var(W) = \frac{2}{n_{in} + n_{out}} \tag{18.1}$$

where n_{in} and n_{out} are the number of inputs and outputs of the layer respectively.

In PyTorch, Xavier initialization can be done using:

(a) Gaussian or normal distribution: `torch.nn.init.xavier_normal_(tensor, gain=1)`

(b) Uniform distribution: `torch.nn.init.xavier_uniform_(tensor, gain=1)`

18.2.2 L2 Regularization

L2 regularisation can be implemented by penalizing the squared magnitude of all parameters directly in the objective function. That is, for every weight w in the network, we add the term $\frac{1}{2}\lambda w^2$ to the objective function, where λ is the regularization strength. A factor of $\frac{1}{2}$ is used because then the gradient of this term with respect to the parameter w is simply λw instead of $2\lambda w$. The L2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors. This encourages the network to use all of its inputs a little rather than some of its inputs a lot. Lastly, during gradient descent parameter update, using the L2 regularization means that every weight is decayed linearly towards zero, as shown in Figure 18.3.

Alternate names for L2 regularisation are:

(a) Weight decay

(b) Ridge

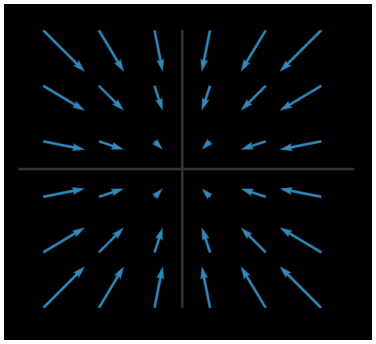


Figure 18.3: Gradient descent parameter update using L2 regularisation

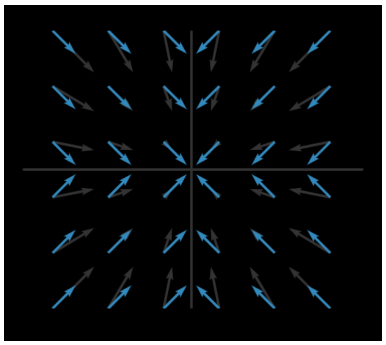


Figure 18.4: Gradient descent parameter update using L1 regularisation

(c) Gaussian prior

In PyTorch, L2 regularisation can be applied using the `torch.optim` module.

18.2.3 L1 Regularization

In L1 regularisation, for each weight w in the network we add the term $\lambda|w|$ to the objective function. Neurons with L1 regularization end up using only a sparse subset of their most important inputs and become nearly invariant to the noisy inputs. Figure 18.4 shows the intuition for the gradient descent parameter update using the L1 regularization. The blue unit vectors are L1 vectors and the grey vectors are the corresponding L2 vectors.

Alternate names for L1 regularisation are:

(a) LASSO (Least Absolute Shrinkage Selector Operator)

(b) Laplacian prior

(c) Sparsity prior

In PyTorch, L1 regularisation can be applied using the `torch.optim` module.

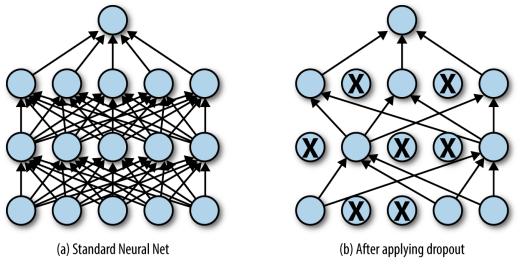


Figure 18.5: Dropout Regularization

18.2.4 Dropout

: Dropout is a regularization technique which is used to prevent overfitting. The technique involves temporarily dropping nodes (hidden and input) along with all its incoming and outgoing connections. During training, dropout removes a unit from the network with probability p(Figure 6.8). For the input nodes, however, the optimal probability of retention is usually closer to 1. At test time, the node is always present and the weights are multiplied by p. This is important because the neurons must have the same output as they had during training time (in expectation). p can be considered to be a hyperparameter.

During training, dropout can be interpreted as sampling a neural network within the full neural network and only updating the parameters of the sampled network based on the input data. However, the exponential number of possible sampled networks are not independent because they share the parameters. During testing there is no dropout applied, with the interpretation of evaluating an averaged prediction across the exponentially-sized ensemble of all sub-networks.

In PyTorch, dropout can be applied using `torch.nn.Dropout(rate=0.5)`.

Several variants are dropout are also available such as:

(a) `torch.nn.Dropout2d(rate=0.5)`

(b) `torch.nn.Dropout3d(rate=0.5)`

(c) `torch.nn.AlphaDropout(rate=0.5)`

18.2.5 Early stopping

: "All standard neural network architectures such as the fully connected multi-layer perceptron are prone to overfitting: While the network seems to get better and better, i.e., the error on the training set decreases, at some point during training it actually begins to get worse again, i.e., the error on unseen examples increases."L. [1998] If we are using an iterative training algorithm such as gradient descent, we can use early stopping for fighting overfitting. The training loss and validation loss decrease with number of epochs in a neural network. However, at a certain point, the validation loss begins to increase while the training loss decreases (See Figure 18.6). We need to stop training at this point as the network starts overfitting here. In each epoch, we check if the validation loss is decreasing; if it is, we save the updated model parameters. At the point where it starts increasing we stop training.

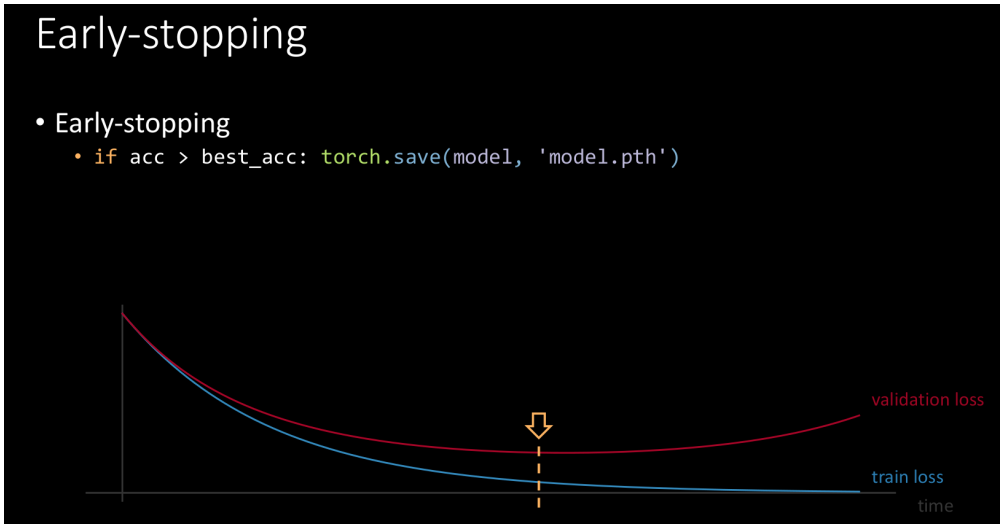


Figure 18.6: Early Stopping

18.2.6 Batch normalization

: Batch normalization reduces the dependence of gradients on the scale of the parameters or on their initial values (reduces internal covariate shift) by fixing the means and variances of layer inputs. It also allows each layer to learn independently of other layers. In addition to this, it also acts as a regularizer which helps reduce overfitting. This is explained by the authors as - "When training with Batch Normalization, a training example is seen in conjunction with other examples in the mini-batch, and the training network no longer producing deterministic values for a given training example. In our experiments, we found this effect to be advantageous to the generalization of the network." (Ioffe and Szegedy [2015])

Pytorch provides the following normalization layers :

(a) `torch.nn.BatchNorm1d(num_features)`

(b) `torch.nn.BatchNorm2d(num_features)`

(c) `torch.nn.BatchNorm3d(num_features)`

18.2.7 Data augmentation

: Another way to fight overfitting is to increase the size of dataset by generating synthetic data. We do this by augmenting the images in dataset using techniques like changing the brightness, cropping, rotating, flipping, scaling and shifting. This helps increase the amount of relevant data and also helps the network generalize better. Adding these synthetically generated images to our training data makes the network robust and invariant to changes. This technique is quite effective in tasks like object detection and speech recognition.

Pytorch has the following transformation functions for data augmentation:

- `torchvision.transforms.CenterCrop(size)`

- `torchvision.transforms.FiveCrop(size)`
- `torchvision.transforms.ColorJitter(brightness, contrast, saturation, hue)`
- `torchvision.transforms.Grayscale(num_output_channels=1)`
- `torchvision.transforms.RandomVerticalFlip(p=0.5)`
- `torchvision.transforms.RandomAffine(degrees, translate, scale, shear)`
- `torchvision.transforms.RandomCrop(size, padding, pad_if_needed, fill)`
- `torchvision.transforms.RandomGrayscale(p=0.1)`
- `torchvision.transforms.RandomHorizontalFlip(p=0.5)`
- `torchvision.transforms.RandomRotation(degrees)`

18.2.8 Transfer learning (TL) and Fine Tuning (FT)

: "Transfer learning and domain adaptation refer to the situation where what has been learned in one setting (i.e., distribution P 1) is exploited to improve generalization in another setting (say distribution P2)." (Goodfellow et al. [2016]) Without transfer learning we would not be able to train networks for tasks with very few examples. We use a model pre-trained on some other related task(Task 1) and use it's knowledge of Task 1 to improve generalization on Task 2. This is done by using the neural network with trained weights and replacing the top layers and retraining these layers. The number of layer replaced depends on the amount of data we have for our task. The type of transfer learning used depends on size of new dataset and its similarity to the original dataset.

- Few data similar to train : If the new dataset is small then we don't fine tune the model as it would lead to overfitting. Since the data is similar to training data, we don't change layers which detect higher level features. We just change the top layer (classifier) and retrain.
- Lots data similar to train: We can fine-tune the entire network since we have a lot of data.
- Few data different from training data : Since the new dataset is different from the training data, the layers which detect higher level features cannot be reused. These layers may contain dataset specific features. Since our dataset is small, we do transfer learning (retrain the classifier) but we start early. The classifier is trained using activation from an earlier layer.
- Lots data different from training data: Since we have a lot of data, we can just train the network and not use transfer learning. We just use the pretrained weights to initialize the network and then train the entire network.

While doing fine tuning we can use diversified learning rates where we train the classifier on top but the other layers don't change much.

Chapter 19

Unsupervised Learning

19.1 Introduction

19.1.1 Machine Learning and the Cake Analogy

Machine learning techniques such as supervised learning (which only predicts human-provided labels), and reinforcement learning (which only predicts a value function), are too narrow to create human-level intelligent machines. Unsupervised learning (or self-supervised learning to be exact), however, with its millions of bits of information per sample, can be used to train highly complex machines without human supervision.

Like supervised learning, self-supervised learning learns a function from pairs of inputs and outputs. However, instead of having annotators manually label the data, self-supervised learning automatically generates labels by extracting weak annotation information from the input data and predicting the rest. In this way the model can independently learn semantic feature representations of data, which can be further used in other tasks.

There are some examples where supervised learning is not suitable and we can use unsupervised learning techniques:

- **Machine translation for any given language pair**: There are about 5k-7k languages, and we cannot do a supervised problem of learning n^2 language-to-language pair translators. A unsupervised learning way to do this is using a encoder-decoder structure we encode any language into a intermediate representation space (n encoder) and decode from this internal representation to the chosen language (n decoder).
- **Learning with very small amount of data**: There is a lot of situation that we don't have a lot of labeled data. People want to detect rare occurrences of things. Train a system to detect rare diseases in medical images or train the automatic driving systems where you cannot to exhaust every cases.
- **Combining deep learning with reasoning**: In some case we need to train systems to reason, not just perceive. Human have a pretty good model that predicts what may happen. So we can know the result without trying it. This model is acquired by observing which is a form of unsupervised learning.

Today's AI technologies can easily classify images and recognize voices, but cannot perform tasks such as reasoning the relationship between different objects or predicting humans' movements. That is where unsupervised learning can fill the blanks. The reason we need unsupervised learning is the fact that the number of the amount of the data we ask machine to predict is sort of different. These different paradigms of learning are very different.

- In the reinforcement learning, you ask machine to predict a reward(cost) of a particular sequence of action. The system just gets a reward instead of much information in the end. It is a very sparse feedback, which means the number of trials needed is very large because in every trial you only give it a little information.
- In supervised learning, you give $10-10,000$ bits per sample. When doing ImageNet classification, for example, there is $1,000$ categories. The information is 10 bits.
- In unsupervised learning, much more information has be used.

There is a analogy of machine learning and the cake: almost everything we learn we learned in unsupervised learning. We learn a little bit through supervised learning, and we learn a tiny bit through reinforcement learning.

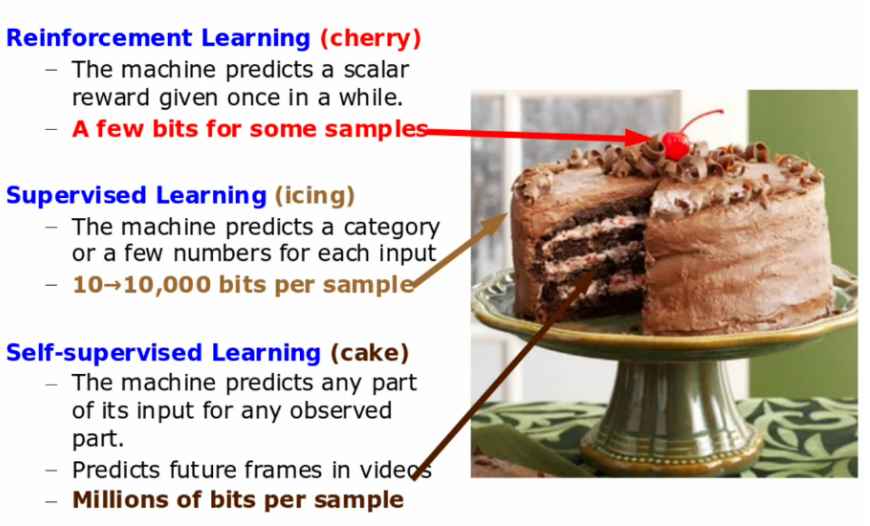


Figure 19.1: Cake Analogy

19.1.2 How do we do unsupervised learning?

Prediction is kind of a reconstruction. For example, you can train a system to predict the end of a video from the beginning. But you can also do it in the other direction(given the end, predict what happened before). Completing the missing information, filling the blanks, that's the purpose of unsupervised learning.

So there is a problem: if I ask you to predict the future of a video with given video clips, many plausible things can happen. There is no single correct answer for this. The energy based learning

will be useful. We need to construct a predictor that can predict multiple things, not just one single future prediction. The world is not entirely predictable. There is a lot of uncertainty. The question is, how do construct machine to predict things. The immediate answer is we can predict a distribution instead of predicting a single point.(e.g. Put a pen on table and let it go. The pen falls in a different direction every time. The directions is in a distribution) If you want to predict a frame of video, there is a distribution of images. You will have to parameterize the distribution of images. The basic possible to parameterize the distribution. We don't have density models or condition models of natural images. The main issue is that we don't know how to represent normalized distributions. Giving a quality of goodness or badness to a particular image is maybe possible, but training this into a distribution is very hard. That is one reason for introducing energy based models.

19.2 Energy-based Unsupervised Learning

19.2.1 Introduction

The way to make multiple predictions, or to build a model makes multiple predictions is through the latent variable. Imagine that for a predictor we are going to train, the x are the frames of the video tape, y is the feature of that particular video tape, and we will have an extra input z to the function. What we would like to have is training the system in such a way that, when the latent variable varies over a particular set, the output varies over the set of all plausible predictions. For a particular z, the predicted next frame y' and the actual y observed, by tweaking the latent variable, the model is to predict the actually observed y.

The idea of latent variable model is that, we have a latent variable of which value is not known, but through optimization, you are able to find the value of it that makes the best prediction.

What to do here is to measure the distance, try to figure out what is the value z within the set, that would minimize the distance between the prediction y' and the y observed. It may not actually be equal, so you will train the network to make these two as closet as possible, for the particular z.

19.2.2 Shape the Energy Function

The energy-based unsupervised learning is identical to energy-based models, except here are Y_1 and Y_2 instead of X and Y. We train the network to figure out the structure of Y, where the energy function will takes low values of the data points nearby and higher values that aside. Capturing those dependencies through the energy function allows you to predict any subset of variables from any other subset of variables.

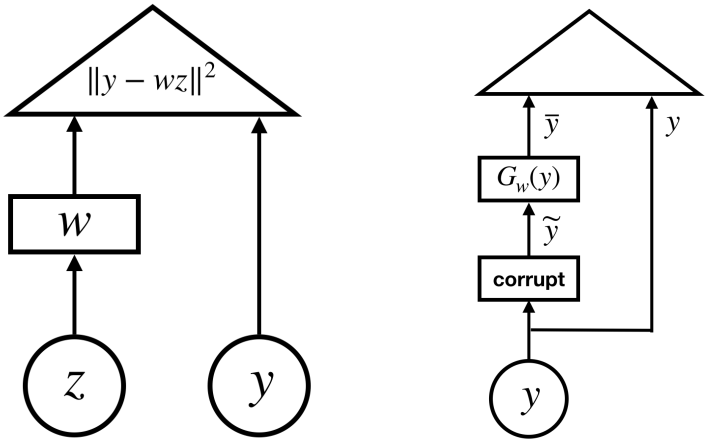
Making energy low on the training samples is easy, which can be achieved just by showing the training data and tweaking the parameters to make energy low on them. The most tricky part is how to ensure the energy is higher outside the training data. There are several ways to do that as shown below.

1. **Constant and Constraint**: The volume of the space that your energy function can give low values to is limited or constant. For example, normalized probability model satisfy this condition, because if at one point the probability goes up, there would be another point that goes down to maintain the normalized distribution. Since the whole volume of probability is constant which equal to 1, the volume of low energy stuff is constant as well.

(a) **Gaussian Mixture Model**.

(b) **Principal Component Analysis**: There is one-hot vector z multiplied by matrix, and goes into euclidean distance, let $E(y, w) = \|y - wz\|^2$. Compute $f(y) = \arg\min_z E(y, z)$, select w that is a prototype closest to y by inferring latent variable z. See Figure 19.2a for an illustration.

(c) **K-means**: The energy function has k valleys, not more than k, the volumes is bounded to constant once k is decided.



(a) Principal Component Analysis (b) Denoising Autoencoder

Figure 19.2: Shape the Energy Function

2. **Push Down and Push Up**: Here are two methods that push down energy of data points while push up elsewhere or on chosen locations, both invented by physicists.

   (a) **Monte-Carlo Sampling**: Replace integral with discrete sum, draw samples and take average.

   (b) **Variational Approximation**: Replace $P(y, w)$ distribution with another distribution Q that is easier to compute (e.g. Gaussian), and minimize the Kullback–Leibler divergence between P and Q.

3. **Minimize Gradient and Maximize Curvature**: It is impractical. We need to make energy flat around data points, which means the derivative being zero, and the second derivative as large as possible. In other words, "Compute the gradients with respect to parameters of the sum of the diagonal terms which is the trace of the Hessian function with respect to the inputs."

4. **Denoising Autoencoder**: This method is useful. How it works is we have an input y and corrupt it with noise, then pass the corrupted y through a parameterized function G, eventually the energy function compares the uncorrupted y and $G(\bar{y})$, gives the distance $\bar{y}$ is away from the manifold which is also the reconstruction error. The noise $\bar{y}$ will have high

energy. There are two problems with this method, the first one is that it is hard to know what noise to add. There are too many ways to corrupt an input, you may not be able to cover all that expand the entire high-dimensional space outside your manifold. The second one it that there is no guarantee the model is going to learn exactly what you taught, for instance, the catastrophic scenario is that the points reconstructed go in circle. See Figure 19.2b above for an illustration.

Chapter 20

Unsupervised Learning

20.1 Energy-based Unsupervised Learning

At a high level, many unsupervised models can be viewed as a scalar-valued energy function $E(Y)$ that operates on input data vector Y. The energy function is subject to learning. Training an unsupervised model consists in searching for an energy function within a family $E(Y, W), W \in \mathcal{W}$ indexed by a parameter W. Both $E(Y)$ and $E(Y, W)$ are designed to produce low energy values when Y is similar to the training data and high energy values when Y is dissimilar to the training data.

Energy-based models have been used to capture dependencies over variables by defining an energy function. The energy function associates each configuration of the variables with a scalar energy value. Lower energy values should be assigned to more likely or plausible configurations and conversely higher values to others. Figure 20.1 shows an example of samples in the manifold, where

$$Y_2 = Y_1^2 \tag{20.1}$$

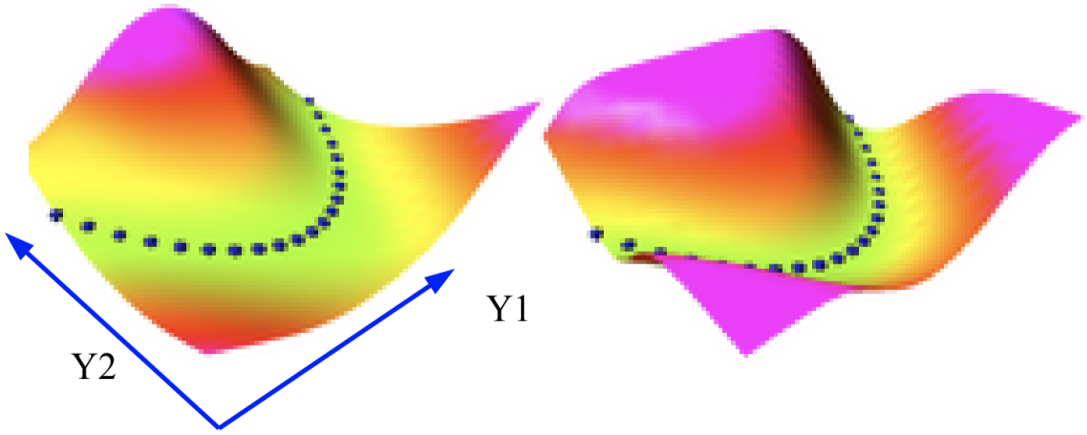


Figure 20.1: Example: The samples live in the manifold

Making the energy low on the samples is easy (through back-propagation). But how do we make it high everywhere else? Pulling up the energies of unobserved points in high dimensional spaces is often very difficult and even intractable. Probabilistic models use a particular method for pulling up the energy of unobserved points that turn out to be very inefficient in many cases.

20.1.1 Strategies to Shape the Energy Function

Build the machine so that the volume of low energy stuff is constant (PCA, K-means, GMM, square ICA)

When it is known that the volume in the y space of the machine can give limited energy (or maybe just constant). Normalized probabilistic models are of this type, if we increase the probability at one point, we have to decrease it at other points. In other words, if we decrease the negative log probability at one point, we have to increase it at other points to maintain the normalization of the distribution. The whole volume of the probability is constant, equal to 1, which means that the volume of the low energy stuff is constant as well. Some normalized models that obeys this condition are Gaussian Mixture Model and PCA. On K-means the volume is not constant, it's bounded, so it does not fully obey this condition.

The architecture of PCA is shown in Figure 20.2. The energy function is:

$$E(y, z) = \|y - wz\|^2 , \tag{20.2}$$

where z belongs to the one hot vector. Then we compute the function

$$F(y) = min_z E(y, z). \tag{20.3}$$

By changing z, we are selecting the column of w that is closest to y by doing this minimization.

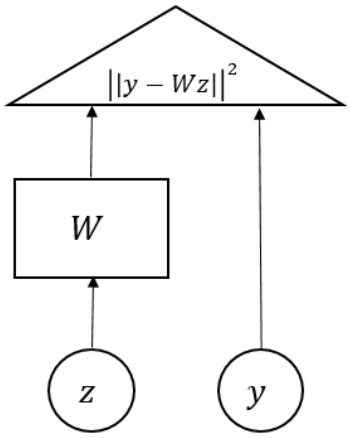


Figure 20.2: The architecture of PCA

The architecture for K-means clustering has no encoder, only a decoder and a reconstruction cost module. The energy function of K-means is shown in Figure 20.3, as y moves away from each prototype, the energy goes quadratically. The volume that can take low energy is limited by k. The only points that are reconstructed with zero energy are the prototypes. Every other point has higher energy.

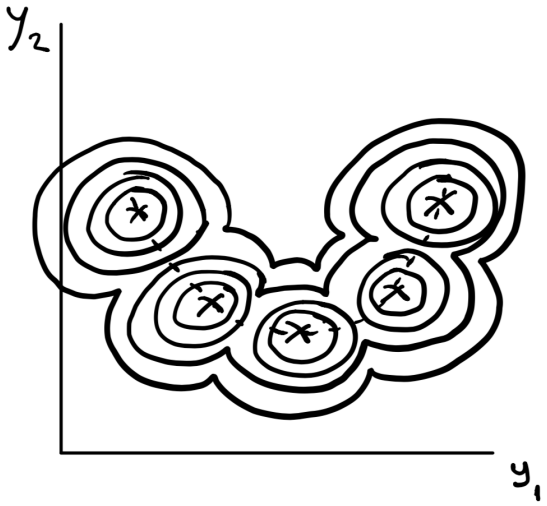


Figure 20.3: Energy function of k-means

In Figure 20.4, the grey level represents energy, dark indicates low energy. Use the PCA as example, PCA avoids flat energy surface by using a code with a lower dimension than the input.

Minimize the distance of every point with the projection, in this example is basically fitting a line to these points. The volume that can take low energy in PCA is limited, only a linear subspace of the input can have low energy, everything outside the subspace will have higher energy.

PCA:

$$E(y,z) = \left\|W^\top WY - Y\right\|^2 \tag{20.4}$$

K-Means

$$E(Y) = min_z \sum_i \|Y - W_i Z_i\|^2 \tag{20.5}$$

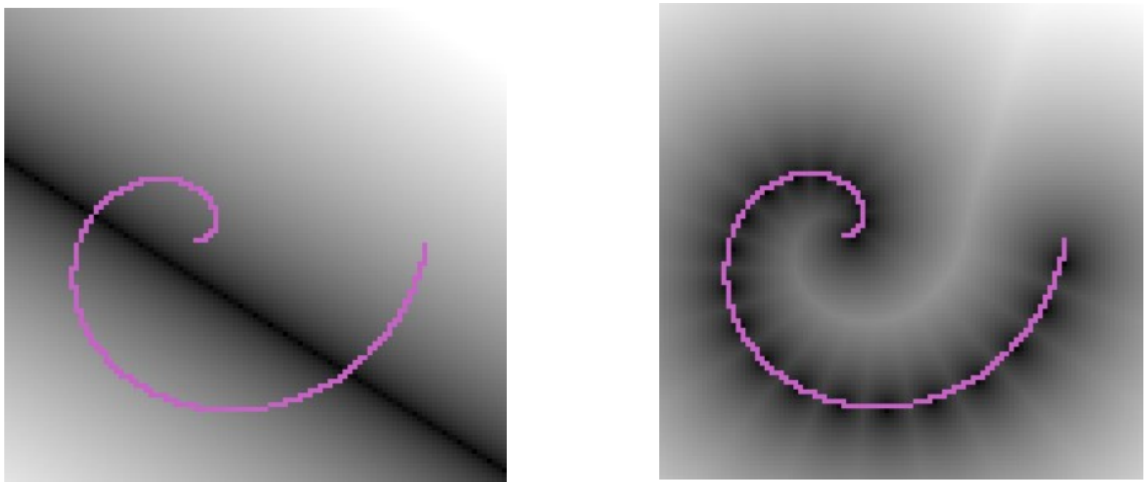


Figure 20.4: Left: PCA and Right: K-Means

Push down of the energy of data points, push up everywhere else (Max likelihood (needs tractable partition function))

This what full-fledged probability methods are doing. The probability of a point y is given by Gibbs distribution which is derived from the energy function:

$$P(Y|W) = \frac{e^{-\beta E(W,Y)}}{\int_y e^{-\beta E(W,y)}} \tag{20.6}$$

Where β is an arbitrary positive constant. Training a probabilistic density model from a training dataset is generally performed by finding the W that maximizes the likelihood of the training data under the model$\sum_{i=1}^{p} P(Y^i, W)$. Equivalently, we can minimize a loss function $L(W,T)$ that is proportional to the negative log probability of the data. Here what we want to do is to minimize the negative log likelihood $L(W,T)$ on the training samples.

$$L(W,Y) = \frac{1}{p}\sum_{i=1}^{p} E(Y^i, W) + \frac{1}{\beta} log \int_y e^{-\beta E(y,W)} \tag{20.7}$$

The gradient of $L(W,T)$ is

$$\frac{\partial L(W,Y)}{\partial W} = \frac{1}{p}\sum_{i=1}^{p}\frac{\partial E(Y^i,W)}{\partial W} - \int_y P(y,W)\frac{\partial E(y,W)}{\partial W} \tag{20.8}$$

In other words, minimizing the first term in equation 20.7 with respect to W has the effect of making the energy of observed data points as small as possible, while minimizing the second term has the effect of "pulling up" on the energy of unobserved data points to make it as high as possible, particularly if their energy is low (their probability under the model is high).

Naturally, evaluating the derivative of the second term in equation 20.7 may be intractable when Y is a high dimensional variable and $E(Y,W)$ is a complicated function for which the integral has no analytic solution. The intractable integral is often evaluated through:

- Monte-Carlo sampling, where we are going to replace the integral with discrete sum, where we draw samples from the distribution and then compute the average of the samples.
- Variational approximations, where we replace the distribution P with one that we know how to compute, like Gaussian. So we pick Gaussian which is possible the best approximation to P, in terms of KL Divergence, which measures the divergence between distributions. We replace P by Q, and then compute (17.8), while minimizing the divergence between P and Q at the same time.

Push down of the energy of data points, push up on chosen locations (contrastive divergence, Ratio Matching, Noise Contrastive Estimation, Minimum Probability Flow)

Monte Carlo methods, adversarial training, generative adversarial networks and many more falls under this category.

Minimize the gradient and maximize the curvature around data points (score matching)

The idea is to take the energy function and try to make it flat around data points. In other words, we want the gradient of the energy function in y space (with respect to input y) to be 0, because we want the energy to be minimum. See figure 20.5 . The ideal energy function is the one that take low values around the points and value increases as we move away. In other words we can define score matching as having an energy function that has zero gradient with respect to the input, while also having a second derivative that is as large as possible. However, the problem is that we have to compute the gradient of the trace of the Hessian with respect to the parameters. This idea is *"cute but impractical"*, except for very simple models.

Train a dynamical system so that the dynamics goes to the manifold (denoising auto-encoder)

The idea of denoising auto-encoder is very useful. We start with the raw input y, and corrupt it with some noise, and then run it through some parameterized function $G_w(y)$. Then we take the uncorrupted input and compute the distance with the output from the function $G_w(y)$. By applying corruption and function $G_w(y)$, we are supposed to reconstruct the original y with its prediction $\bar{y}$. Then we minimize the reconstruction error, which is the square of distance between $\bar{y}$ and y. See

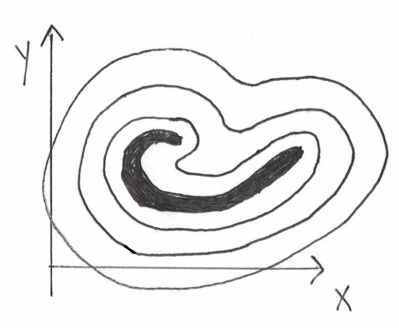


Figure 20.5: Energy Function

20.6. In other words, the function $G_w(y)$ maps the point outside the manifold to the point inside the manifold. At test time, we remove the corrupt part and give a y without corruption.

The energy-based view of denosing auto-encoder is that when we train the function to map the points outside the manifold to the points on the manifold, we train it to compute the energy, which is the reconstruction error.

Two slight problems remain:

- We don't know what noise, or what types of corruptions make sense.
- There are many ways that one can corrupt an input in the high dimensional space. We may not be able to cover all possible ways to corrupt an input that can span the high dimensional space. Besides, there is no guarantee that the model learns exactly what we teach it. It may not be able transform all the points to the manifold.

Use a regularizer that limits the volume of space that has low energy

This will be discussed in more detail on April 29.

If $E(Y) = ||Y - G(Y)||^2$, make $G(Y)$ as "constant" as possible.(Contracting auto-encoder, saturating auto-encoder)

Contracting auto-encoder comes from Yoshua Bengio's lab. Basically, it's still in the architecture of auto-encoder in figure 20.7.

The energy of y is

$$E(y) = ||y - G(y)||^2,$$

and the loss function is

$$L = \sum_i E(y^i) + R(w),$$

where the regularization term $R(w)$ is the square Frobenius norm of Jacobian of the hidden layer representation with respect to input values. This encourages a constant hidden representation except around training samples where it is counteracted by the reconstruction term.

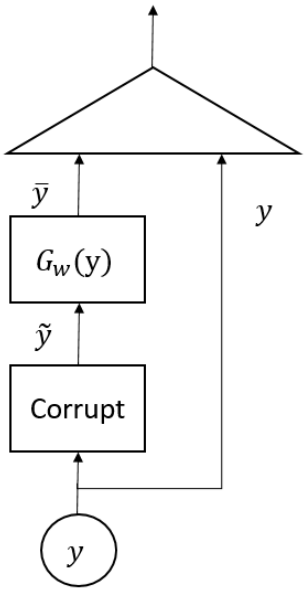


Figure 20.6: The architecture of denoising auto-encoder

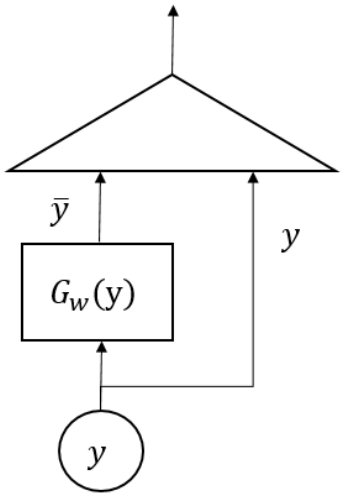


Figure 20.7: The architecture of auto-encoder

Chapter 21

Bayesian Neural Networks

21.1 Motivation of estimating a predictive distribution

Most of modern neural network models produce point estimation, even though at last phase of image classification models we get confidence scores of different categories like cat and dogs. With complexity/uncertainties happening all the time around the world, it is not always reasonable to get a single output result. Sometimes we cannot tell whether the new model is making sensible predictions or just guessing at random. In this situation, we may wish to get a predictive distribution of results instead of a point estimation.

21.2 Motivation of caring about uncertainty

Given several pictures of cats and dogs, then being asked to classify a new cat photo, we should return a prediction with rather high confidence. But if we are given a photo of an ostrich and force our hand to decide if it is a cat or dog, we should return a prediction with very low confidence. It is quite important to say whether we are confident about our estimation or not. Another example is predicting physics simulator. In physics, you may have simulators for predicting the trajectory of some particles or accelerators. But, then this simulation takes forever. In this case, you can use neural networks to speed up computations and getting approximate values. The network is trained over many iterations using data provided by simulators. However, if inputs are far from trained data, a prediction is made with higher uncertainty. Furthermore, this sort of information lies at the foundations of artificial intelligence as well. For example, a Roomba vacuum needs to learn about its environment (e.g. living room) based on its actions (rolling around in different directions). It can decide to go forward and might bump into a wall. One should enable reward mechanism to encourage it when it learns to avoid the wall or penalize it when it crahses into a sofa. This setting is known as reinforcement learning. The Roomba is requried to explore its environment looking for these rewards, and that's where uncertainty comes into play. The Roomba will try to minimise its uncertainty about different actions - and trade-off between this exploration, and exploitation of what it already knows. You can trace to Karpathy's (https://cs.stanford.edu/people/karpathy/convnetjs/demo/rldemo.html) interactive demo to learn

into this Roomba play.

21.3 Get predictive distribution using dropout

Take any network trained with dropout and some input x^*. We are looking for the expected model output given our input - the predictive mean $\mathbb{E}(y^*)$ and how much the model is confident in its prediction - the predictive variance $Var(y^*)$. The neural network model with enabled dropout in both train and test phases will be equivalent to Gaussian process approximation. In the following section we will walk through the process of enabling dropout and getting the estimation of predictive distribution.
First, define a prior length-scale l. This captures our belief over the function frequency. A short length-scale l corresponds to high frequency data, and a long length-scale corresponds to low frequency data.Take the length-scale squared, and divide it by the weight decay. We then scale the result by half the dropout probability over the number of data points. Mathematically this result in a Gaussian process precision $\tau = \frac{l^2 p}{2N\lambda}$ we mentioned above. Note that p here is the probability of the units not being dropped - in most implementations p_{code} is defined as the probability of the units to be dropped, thus $p := 1 - p_{code}$ should be used when calculating τ. Next, simulate a network output with input x^*, treating dropout as if we were using it during training phase. This means enabling dropout also in test phase. Repeat this T times with different units dropped every time, and collect the results $\hat{y_t}^*(x^*)$. These are empirical samples from our approximate predictive posterior. We can get an empirical estimator for the predictive mean of our approximate posterior as well as the predictive variance (our uncertainty) from these samples. Mathematically, averaging forward passes through the network is equivalent to Monte Carlo integration over a Gaussian process posterior approximation. The derivation is provided in Gal and Ghahramani [2016]. The process of applying dropout into the neural network at both training and test phases is thus called Monte-Carlo dropout (MC dropout). We simply follow these two equations:

$$\mathbb{E}(y^*) \approx \frac{1}{T}\sum_{t=1}^{T} \hat{y_t}^*(x^*) \tag{21.1}$$

$$Var(y^*) \approx \tau^{-1} I_D + \frac{1}{T}\sum_{t=1}^{T} \hat{y_t}^*(x^*)^T \hat{y_t}^*(x^*) - \mathbb{E}(y^*)^T \mathbb{E}(y^*) \tag{21.2}$$

Eq. 17.1 was introduced as model averaging and it was explained that scaling the weights at test time without dropout gives a reasonable approximation to this equation. Eq. 17.2 is simply the sample variance of T forward passes through the network plus the inverse model precision. Note that the vectors above are row vectors and that the products are outer-products.
Implementation in python code to get the predictive mean and uncertainty is easy:

```
probs = []
for _ in xrange(T):
    probs += [model.output_probs(input_x)]
predictive_mean = numpy.mean(prob, axis=0)
predictive_variance = numpy.var(prob, axis=0)
tau = l**2 * (1 - model.p) / (2 * N * model.weight_decay)
predictive_variance += tau**-1
```

Figure 21.1: Python code to obtain predictive mean and uncertainty from dropout

21.4 Understanding MC dropout Models

21.4.1 Regression

To see what the uncertainty looks like where points are far away from training data, we can visualize it using a simple one dimensional regression model with a subset of the atmospheric CO_2 concentrations dataset. This dataset was derived from situ air samples collected at Mauna Loa Observatory, Hawaii, since 1958. The dataset consists of about 200 data points and has been centered and normalized. Fig. 21.2 shows the raw data, and 21.3 shows the processed training data (in red, left of the dashed blue line) and a missing section to the right of the dashed blue line. A neural network with 5 hidden layers, 1024 units in each layer, and ReLU non-linearities, and dropout with probability 0.1 after each weight layer, was fitted to the data.

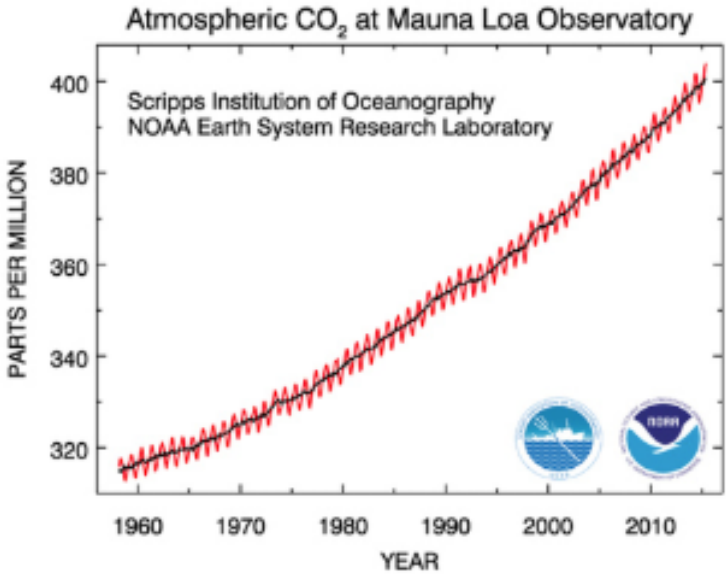


Figure 21.2: CO_2 dataset before pre-processing

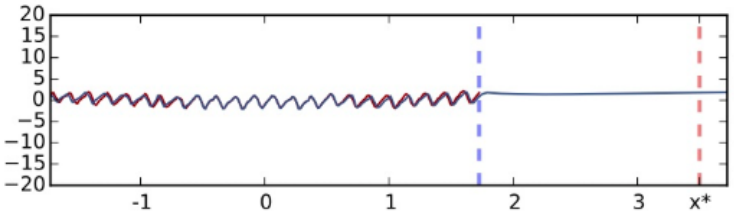


Figure 21.3: Standard dropout network without using uncertainty information

The point marked with a dashed red line is a point far away from the training data. As can be seen, standard dropout confidently predicts a clearly insensible value for the point, as the function is obviously periodic. Therefore, it is really hard to tell whether the model can be trusted or not. However, if we add the uncertainty information introduced above to the exactly same network (we don't need to re-train the model, just perform predictions in a different way), we can get the revealing information shown in Fig. 21.4.

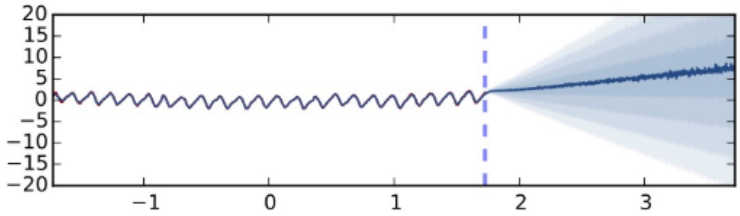


Figure 21.4: Same dropout network performing predictions using uncertainty information

As can be seen, the model does capture a large amount of uncertainty information about the far away test point. This uncertainty information is very similar with that of a Gaussian process model with a squared exponential covariance function (as shown in Fig. 21.5). Though the estimate is different, the uncertainty still increases as the test point goes farther from the training data. It's not surprising that the uncertainty looks different, since the ReLU non-linearity corresponds to a different Gaussian process covariance function. If we change the non-linearity to a TanH non-linearity, the result looks as shown in Fig. 21.6.

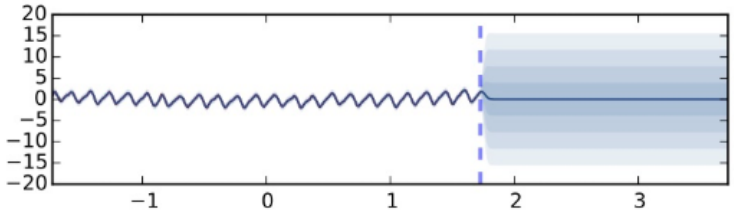


Figure 21.5: Gaussian process with SE covariance function on the same dataset

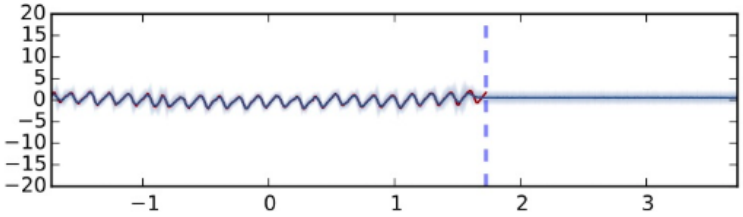


Figure 21.6: Dropout network using uncertainty information and TanH non-linearity

This time it seems that the uncertainty doesn't increase as farther from the training data. This might be because TanH saturates whereas ReLU does not. This non-linearity will not be appropriate for tasks where we are expecting the uncertainty to increase as the test point goes farther away from the training data.

21.4.2 Classification

This section takes an example of image classification using MNIST digits dataset and the popular LeNet convolutional neural network. In this model the prediction is fed into a softmax which gives probabilities for different classes (the 10 digits). However, these probabilities are not enough to see if the model is certain in its prediction or not. This is because the standard model would pass the predictive mean through the softmax rather than the entire distribution.

If we take an idealized binary classification example (as shown in Fig. 21.7). Passing a point estimate of the mean of a function (a TanH function for simplicity, solid line on the left) through a softmax (solid line on the right) results in highly confident extrapolations with x^* (a point far from the training data) classified as Class 1 with probability 1. However, passing the distribution (shaded area on the left) through a softmax (shaded area on the right) reflects classification uncertainty better at points far from the training data. Taking the mean of this distribution passed through the softmax we get Class 1 with probability 0.5 – the model's true prediction.

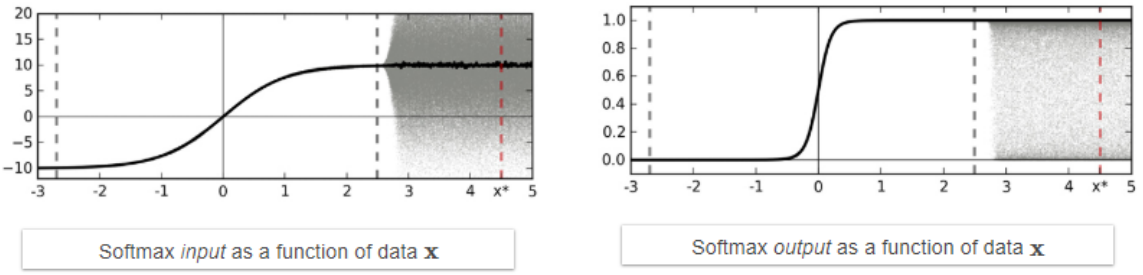


Figure 21.7: A sketch of softmax input and output for an idealized binary classification problem

Then we can try passing the entire distribution through the softmax instead of the mean alone. The samples are simulated through the network and the softmax output is averaged. The convolutional neural network is trained over MNIST with dropout applied after the last inner-product layer (with probability 0.5). The model predictions are evaluated over the following sequence of images (Fig. 21.8), that correspond to some projection in the image space.

Figure 21.8: Image inputs (a rotated digit) to the dropout LeNet network

These images correspond to the X axis in the idealized depiction above. A histogram of 100 samples obtained by simulating forward passes through the dropout LeNet network is shown in Fig. 21.9. As can be seen, Class 7 has low uncertainty for the right-most images. This is because the uncertainty "envelope" of the softmax input for these images is far away from the envelopes of all other classes. In contrast, the uncertainty envelope of Class 5 for the middle images interests the envelopes of some of the other classes (even though its mean is higher) – resulting in large uncertainty for the softmax output. It is important to note that the model uncertainty in the softmax output can be summarized by taking the mean of the distribution. In the idealized example above, this

would result in softmax output 0.5 for point x^* (instead of softmax output 1) and here it will result in a lower softmax output that might result in a different image class. This sort of information can help us in classification tasks: obtaining higher classification accuracies. It also helps us analyze our model and decide whether we have enough data or if the model is specified correctly.

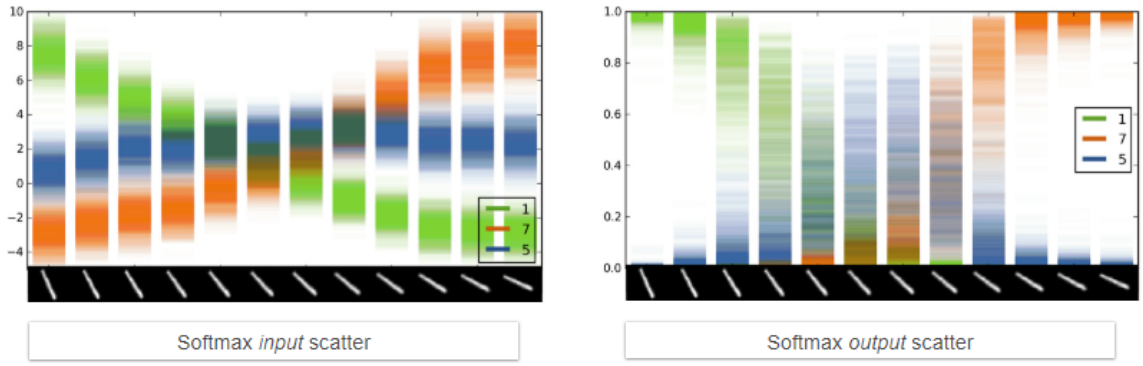


Figure 21.9: A sketch of 100 forward passes of the softmax input and output for dropout LeNet

Part II

Practice

Chapter 22

The Manifold Hypothesis

22.1 Facial Expressions Thought Experiment

Say we have infinitely many pictures of a person making all possible facial expressions. Each image is 2000×1000 pixels and has 3 color channels, so each image is $6,000,000$-dimensional vector.

The set of all images is a small subset of $6,000,000$-dimensional space. What does that subset look like? What is the dimensions of the surface? On a patch of that surface, how many dimensions can you move and still stay on that surface?

That surface is a manifold (roughly speaking, a continuous surface). Moreover, it is limited by the number of degrees of freedom in the human face, which is bounded above by the number of muscle groups a person can control in his face. Ergo that subset of $\mathbb{R}^{6000000}$ is relatively low-dimensional.

This thought experiment illustrates the manifold hypothesis, which postulates that natural data in high dimensional space generally has a low dimensional structure.

Chapter 23

Tensor Transformations

The following sections summarize and visualize how you can transform data represented in matrix form. Input data can be defined as a matrix with the i^{th} row corresponding to the i^{th} data point and each additional columns representing a new dimension of the data. $\boldsymbol{X}$ in the below examples is an input matrix with 1000 data points and two dimensions whose values are standard normally distributed. In the following figures, data points are colored according to their original $x_{N,1}$ dimension violet to yellow for negative to positive values.

$$\boldsymbol{X} = \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \\ x_{3,1} & x_{3,2} \\ \vdots & \vdots \\ x_{1000,1} & x_{1000,2} \end{bmatrix}$$

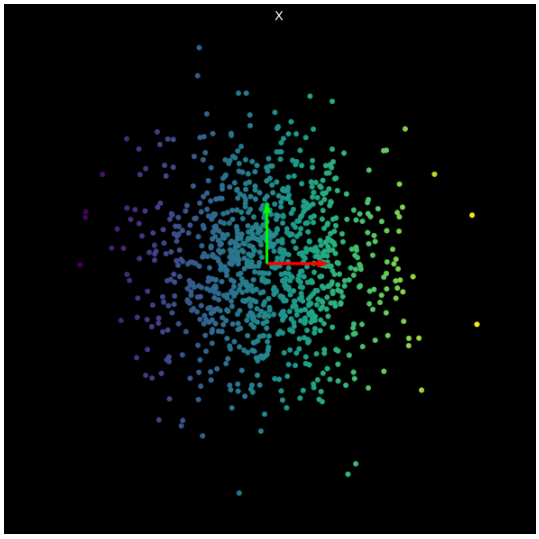


Figure 23.1: Original $\boldsymbol{X}$ Visualized

23.1 Linear Transformations

There are several linear transformation that can be executed on $\boldsymbol{X}$ including:

- Rotation ($\boldsymbol{U}$)
- Scaling (s_1, s_2)
- Reflection ($\boldsymbol{V}$)
- Shearing
- Translation

The product of the first three form weights as shown below.

$$\boldsymbol{W} = \boldsymbol{U} \begin{bmatrix} s_1 & 0 \\ 0 & s_2 \end{bmatrix} \boldsymbol{V}^\top$$

23.1.1 Rotation

During rotation, each point is rotated about the origin by the indicated angle. The below equation populates $\boldsymbol{U}$ based on rotating points by θ counter-clockwise.

$$\boldsymbol{U} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ \sin(\theta) & -\cos(\theta) \end{bmatrix}$$

Keeping scaling and reflection constant, the below figure shows a set up points with different θs

.

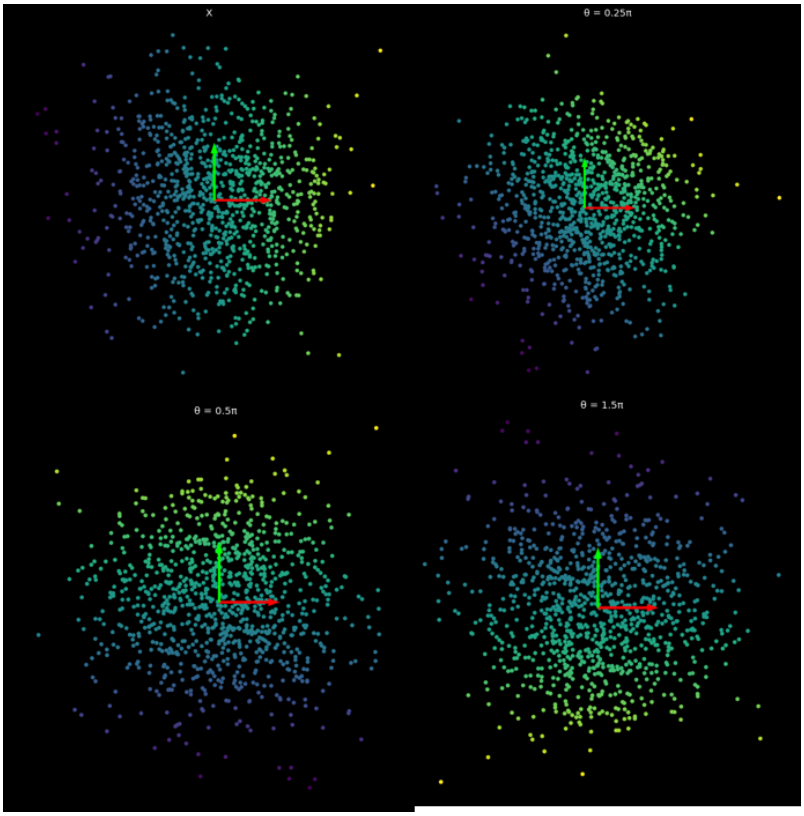


Figure 23.2: Rotation Visualized

23.1.2 Scaling

Scaling the points expands or contracts the points about the origin. This controlled separately for each dimension by s_1 and s_2 .

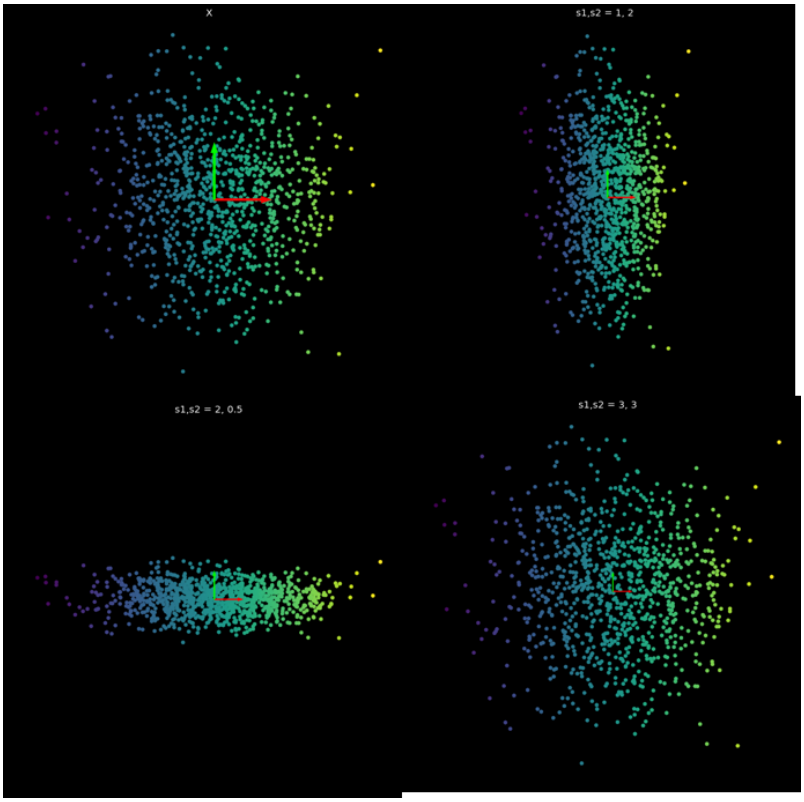


Figure 23.3: Scaling Visualized

23.1.3 Reflection

Reflecting points projects them on the other side of a defined line that crosses the origin. The line that goes from the original point to the projected point is perpendicular to the defined line, and the intersection to the defined line is the midpoint.

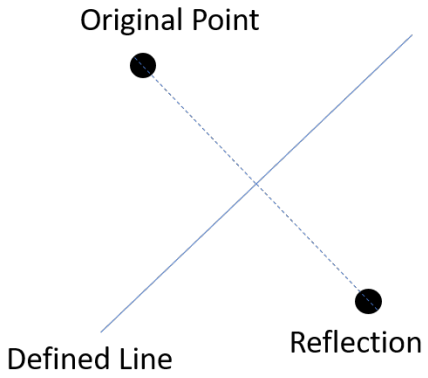


Figure 23.4: Reflection Defined

$\boldsymbol{V}$ can be defined by

$$\boldsymbol{V} = \begin{bmatrix} \cos(2\theta) & \sin(2\theta) \\ \sin(2\theta) & -\cos(2\theta) \end{bmatrix}$$

Where the defined line is $x_2 = \tan(\theta)x_1$

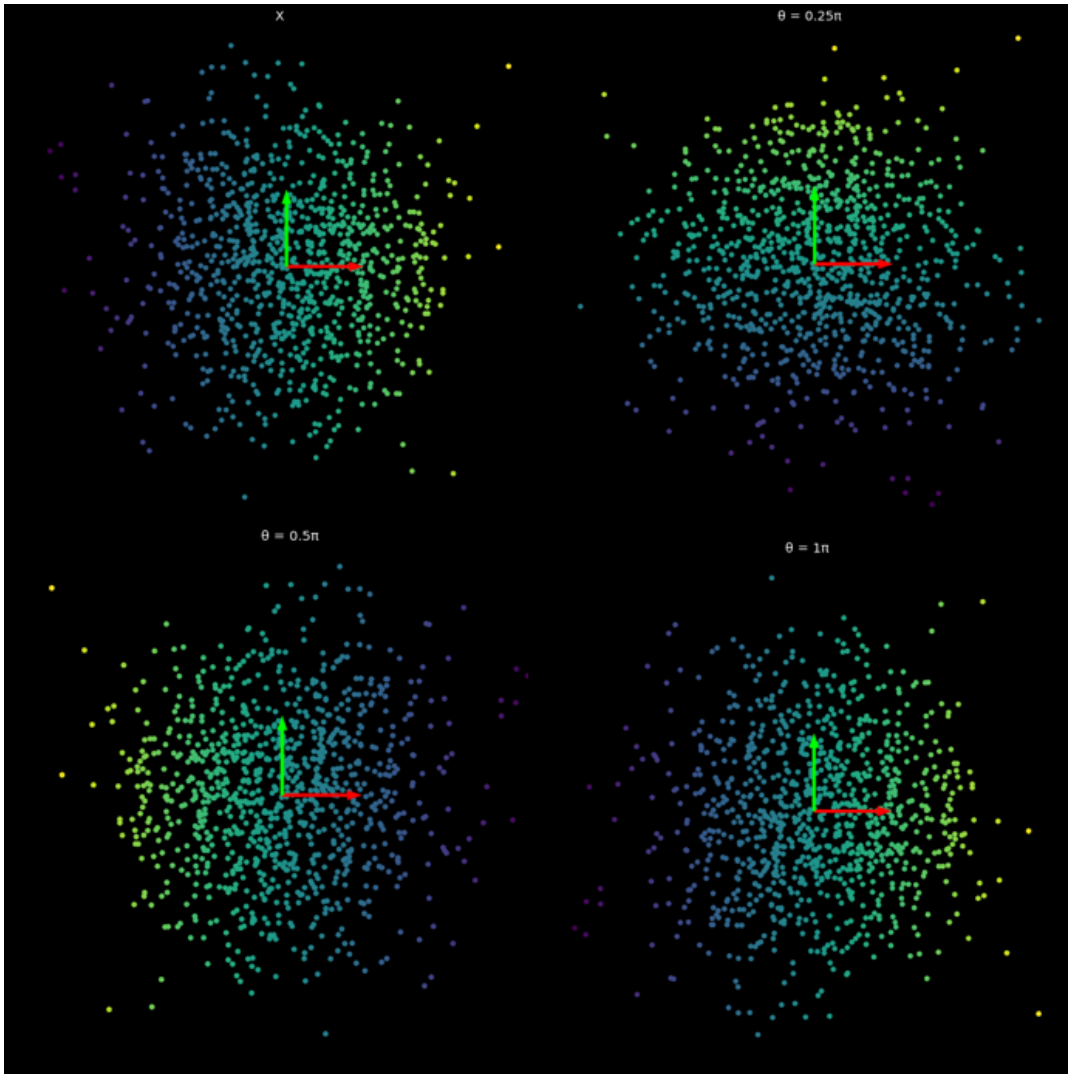


Figure 23.5: Reflection Visualized

23.1.4 Shearing

Shearing points, which is separate from the weight calculation, shifting points in one dimension proportional to their value in the other dimension.

$\boldsymbol{Y} = \boldsymbol{X} \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$ Shift points on the x_1 dimension proportional to the x_2 dimension

$\boldsymbol{Y} = \boldsymbol{X} \begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix}$ Shift points on the x_2 dimension proportional to the x_1 dimension

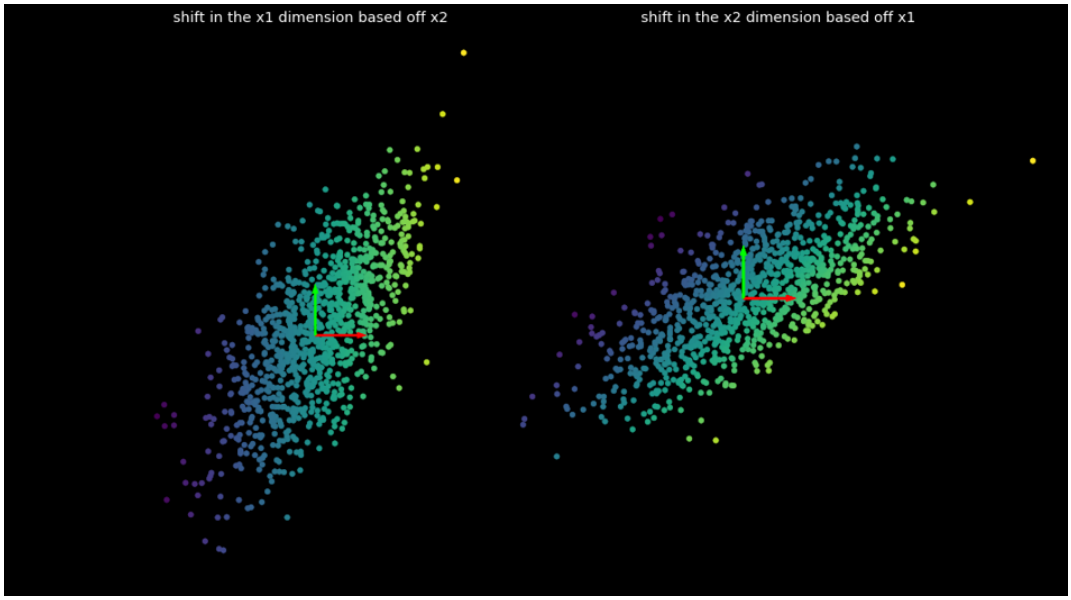


Figure 23.6: Shear Visualized

23.1.5 Translation

The translation of points, which is separate from the weight calculations moves them uniforming in the direction indicated.

$$\boldsymbol{Y} = \boldsymbol{X} + \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ \vdots & \vdots \\ 1 & 1 \end{bmatrix} \begin{bmatrix} t_1 & 0 \\ 0 & t_2 \end{bmatrix}$$

(Shift points left or right by t_1 and up or down by t_2 dimension)

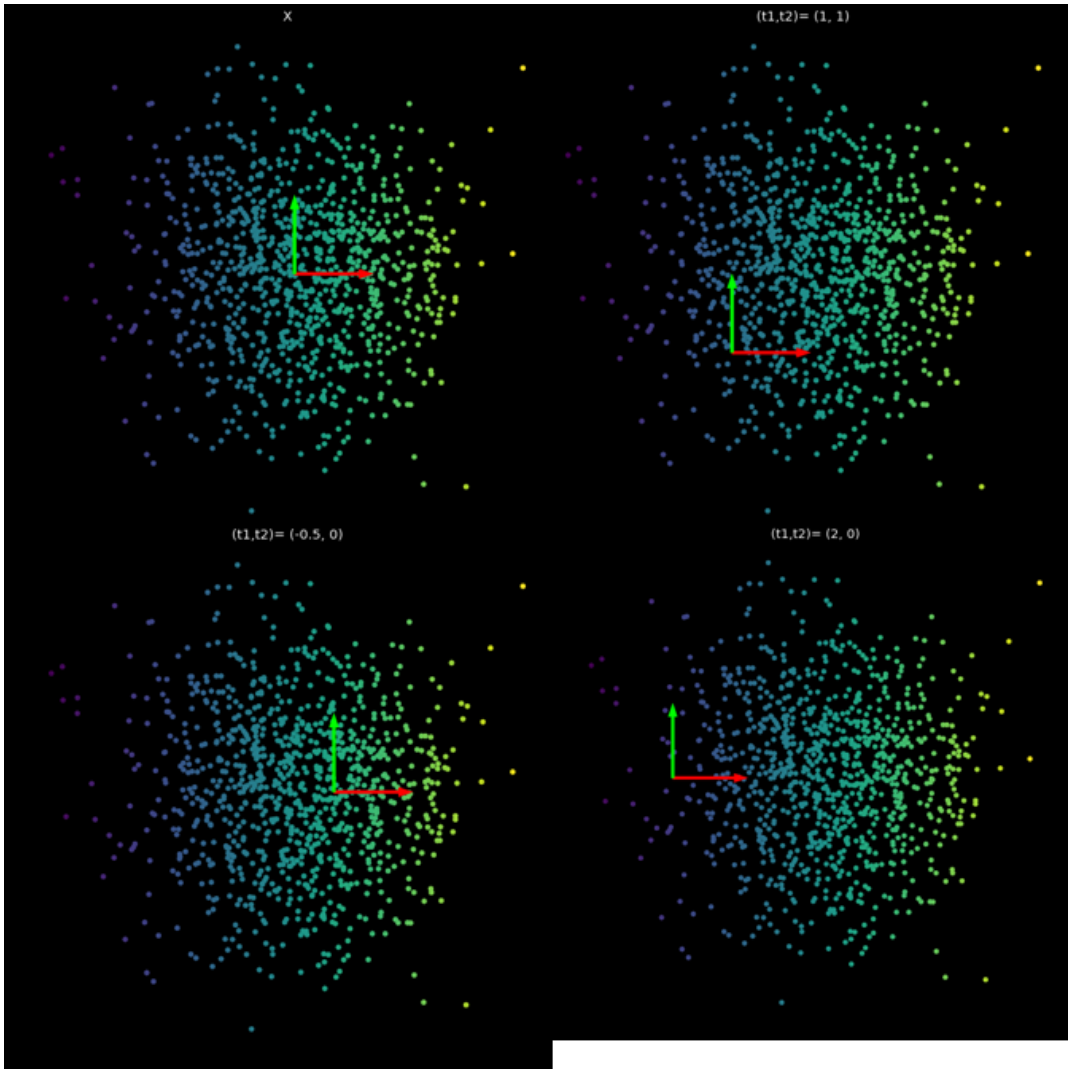


Figure 23.7: Translation Visualized

23.2 Non-Linear Transformations

Linear transformations are capable of altering data in many different ways, but linear transformations cannot curve data. That is where non-linear transformations come in. There are several types of non-linear transformations such as:

- Rectified linear unit (ReLU) - $y = x_1$ if $x_1 > 0$ else 0
- Polynomial- $y = x_1^2$ or $y = x_1 x_2$

- Step - $y = 1$ if $x > 0$ else 0

One of the most common non-linear transformations is the hyperbolic tangent, which in the context of 2D Tensors can be applied as below:

$$f(x) = \tanh(\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} x)$$

This function first stretches out the points using scalar s via scaling, then the hyperbolic tanget squashes the points into a square. The larger s is, the more points end up in the square.

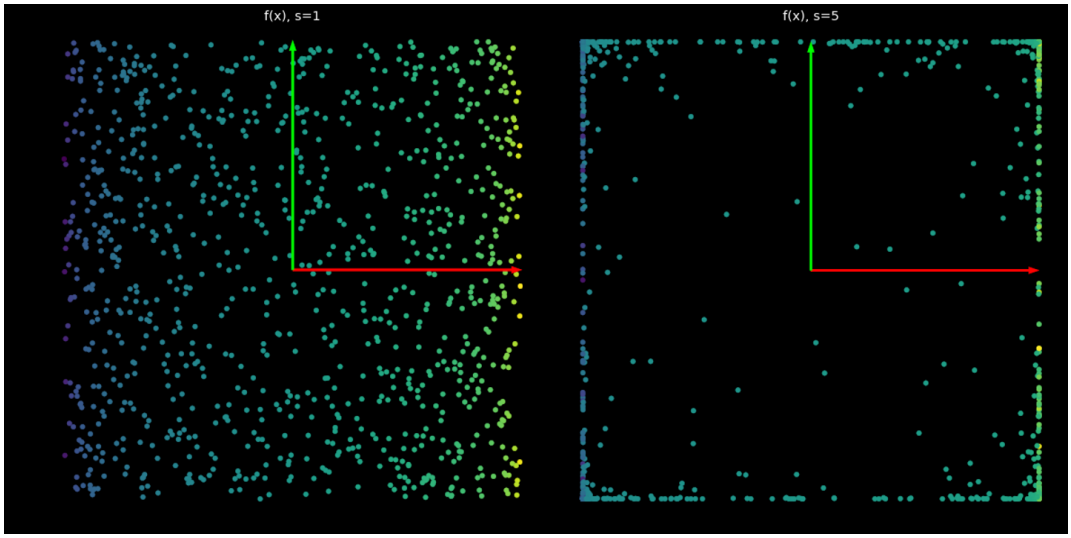


Figure 23.8: Tanh in Isolation

Tanh can create curved surfaces when it is sandwiched in-between two linear transformations in a three layer neural network.

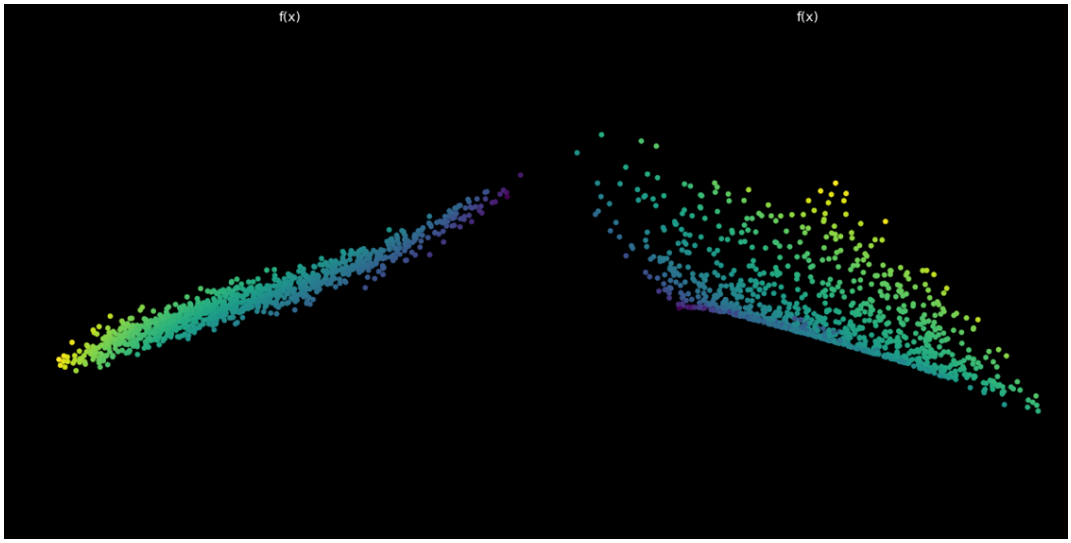


Figure 23.9: Tanh In-Between Two Linear Layers

Chapter 24

Artificial Neural Nets - Supervised Learning: Classification

24.1 Example of Not Linearly Separable Curves

We have three curves defined by functions:

$$X_c(t) = t \begin{pmatrix} \sin(\frac{2\pi}{C}(2t + c - 1)) \\ \cos(\frac{2\pi}{C}(2t + c - 1)) \end{pmatrix}$$

$$0 \leq t \leq 1 \quad c = 1, ..., C$$

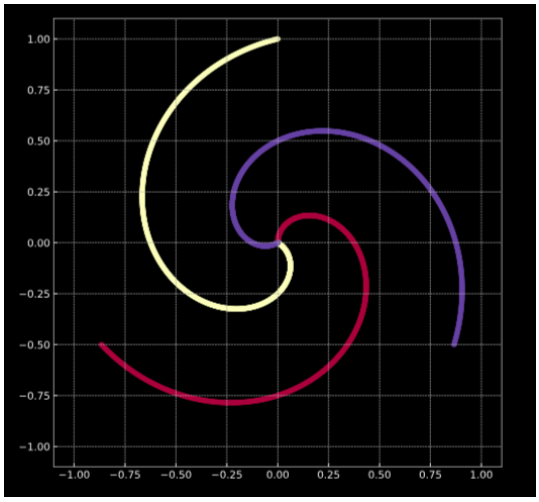


Figure 24.1: 3 not linearly separable parametric curves

If we were to use two linear transformations W_h and W_y transform the data x as so:

$$\boldsymbol{h} = \boldsymbol{W_h x} + \boldsymbol{b_h}$$

$$\hat{\boldsymbol{y}} = \boldsymbol{W_y h} + \boldsymbol{b_y}$$

we would not be able to linearly separate the data points in these curves as shown in figure 24.2. These points are defined by the same function with added noise.

$$X_c(t) = t \begin{pmatrix} sin(\frac{2\pi}{C}(2t + c - 1) + \mathcal{N}(0, \sigma^2)) \\ cos(\frac{2\pi}{C}(2t + c - 1) + \mathcal{N}(0, \sigma^2)) \end{pmatrix}$$

$$0 \leq t \leq 1 \quad c = 1, ..., C$$

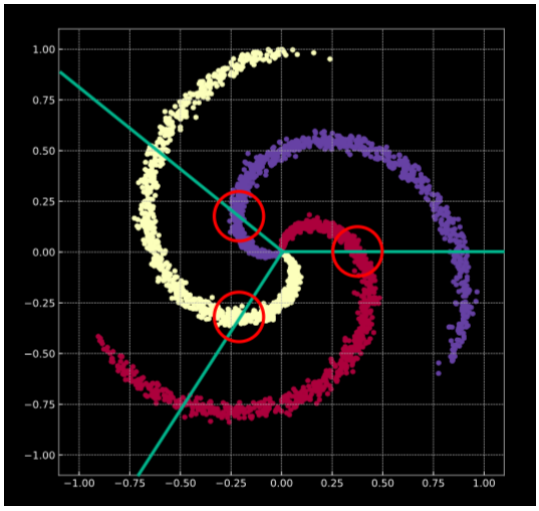


Figure 24.2: These data points of three different classes (represented by color) generated from curves with added noise cannot be separated only with linear functions.

In this case, the best a classifier consisting of solely linear layers can do is to rotate the decision boundaries centered around the origin to predict slightly more than half of the data points correctly as shown in figure 24.3. We are limited to only the linear transformations of rotations, scaling and shearing. We are not able to "warp" the data in the higher dimensional space used in our linear layers to allow for separation in that higher dimension without a non-linear activation function.

To improve the accuracy of the classifier, we introduce the non-linear functions f and g resulting in the 3-layer architecture depicted in figure 24.4. Common selections for these functions include ReLU, tanh, sigmoid and softmax. Finally, the equations that define the network are as follows:

$$\boldsymbol{h} = f(\boldsymbol{W_h x} + \boldsymbol{b_h})$$

$$\hat{\boldsymbol{y}} = g(\boldsymbol{W_y h} + \boldsymbol{b_y})$$

where $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{h} \in \mathbb{R}^d$ and $\hat{\boldsymbol{y}} \in \mathbb{R}^C$. For the current example, the input space is two-dimensional ($n = 2$), the hidden high-dimensional space has $d = 100$ and the output space has $C = 3$ (the number of classes of the original classification problem). The dimension of the hidden space for this

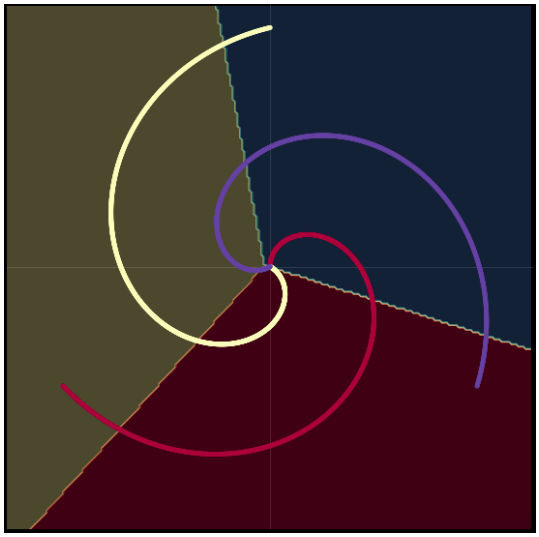


Figure 24.3: 2 linear layers alone cannot create a classifier that performs well for this data

examples satisfies $d >> n$, which spreads the data points making them easier to manipulate and separate than in the original low-dimensional input space.

The dimensions of the weight matrices and bias terms must be selected so as to be consistent with these definitions, thus $\boldsymbol{W_h} \in \mathbb{R}^{d \times n}$, $\boldsymbol{W_y} \in \mathbb{R}^{C \times d}$, $\boldsymbol{b_h} \in \mathbb{R}^d$ and $\boldsymbol{b_y} \in \mathbb{R}^C$.

It is worth noticing that for each non-linearity there is an associated affine transformation, characterized by a rotation $\boldsymbol{W}\boldsymbol{x}$ and a translation $\boldsymbol{b}$.

24.1.1 2 Ways to View Neural Nets

When diagramming or describing a neural network, we begin with the input data at the bottom and then build up with our linear and non-linear transformations to the output at the top. With this structure defined, there are 2 ways to view the behavior of a model (figure 24.5):

1. **Bottom Up**
   Looking up at the original data with decision boundary that has been warped by the model.
2. **Top Down**
   Looking down at data that has been warped by the model.

24.1.2 Output 1-hot encoding

In our previous example, the most straightforward way to map the categorical labels {Red, Green, Blue} into numerical values would be to simply assign a number to each color, *i.e.* {Red=1, Green=2, Blue=3}. However, this approach is not compatible with the softmax function that is typically used for classification problems, which assigns probability-like scores between 0 and 1 to each category.

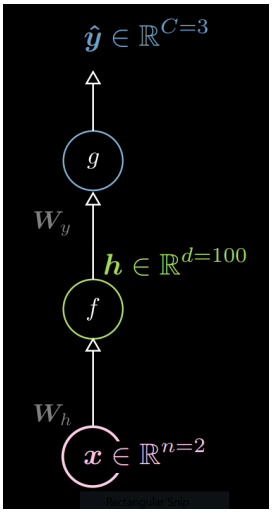


Figure 24.4: Architecture of the 3-layer ANN used for the spiral classification problem, where f and g are non-linear functions.

Furthermore, using natural numbers introduces a notion of distance or proximity that is not really present in the original categorical data. For instance, one could be inclined to think that Blue is more similar to Green than it is to Red because 3 is closer to 2 than it is to 1. The truth is, however, that all three categories are equally dissimilar and this notion of distance does not exist among them.

To avoid this problem and to conform to the output generated by the softmax layer, we encode the labels using a technique known as *1-hot encoding*. For a classification problem with C classes, each class will be mapped to a unit vector in $\mathbb{R}^C$. The vector corresponding to the i-th class in the original problem will have a 1 in its i-th dimension and 0's elsewhere (hence the name 1-hot encoding).

Applying this to the problem at hand we have Red $= (1, 0, 0)$, Green $= (0, 1, 0)$ and Blue $= (0, 0, 1)$.

24.1.3 Neural Network Training: Part 1

The last layer of a neural network trained for a classification problem is usually a softmax function, which is defined as follows:

$$\hat{\boldsymbol{y}}[c] = \text{softmax}(\boldsymbol{l})[c] = \frac{\exp(\boldsymbol{l}[c])}{\sum_{j=1}^{C} \exp(\boldsymbol{l}[j])} \in (0, 1)$$

Where $\boldsymbol{l}$ is the *logits* vector of raw (non-normalized) predictions that the classification model generates, which is then passed to the normalization function, *i.e.* softmax. The notation $[c]$ refers to the c-th element of the vector, which leads us to conclude that $c \in [0, C-1]$ according to the 1-hot encoding described in section 24.1.2 and considering a 0-indexed vector (as it is the default and standard in Pytorch).

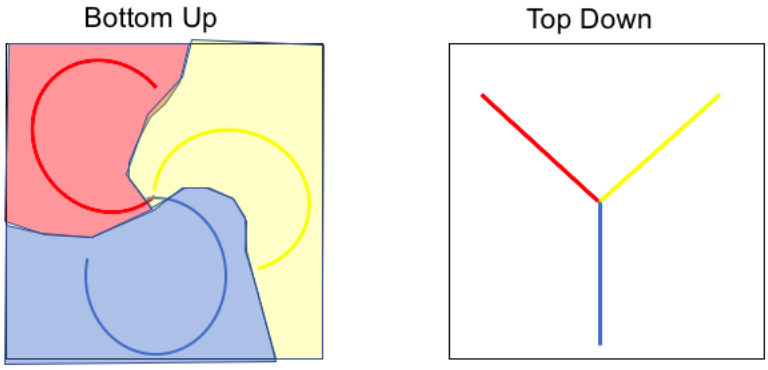


Figure 24.5: Sketches of the two ways of viewing a Neural Net: bottom up (left) and top down (right).

Given the index c in the 1-hot encoding representation of the correct label and the vector of probabilities $\hat{\boldsymbol{y}}$ returned by the model, we can define the *cross-entropy* or *negative log-likelihood* loss as follows:

$$\ell(\hat{\boldsymbol{y}}, c) = -\log(\hat{\boldsymbol{y}}[c])$$
$$\mathcal{L}(\hat{\boldsymbol{Y}}, \boldsymbol{c}) = \frac{1}{m}\sum_{i=1}^{m} \ell(\hat{\boldsymbol{y}}^{(i)}, c^{(i)})$$

Since the softmax produces exponentiation of the logits, we never receive exactly 0 or 1. Instead, 0^+ or 1^- are returned which signify slightly greater than 0 and slightly less than 1. Since the loss is the cross entropy/negative log of the output, if the correct class is predicted the loss will be $-\log(1^-)$, which is 0^+. Similarly, for the other class $-\log(0^+)$ will be $+\infty$.

24.1.4 Neural Network Training: Part 2

Finally, How to train the Neural Network.

$$\Theta \equiv \{W_h, b_h, W_y, b_y\}$$

Here Θ is a conglomeration of all the parameters and it represents one point in the parameter space.

$$\mathcal{J}(\Theta) \equiv \mathcal{L}(\hat{\mathcal{Y}}(\Theta, c) \in R^+$$

$J(\Theta)$ is the loss over the entire data set and is a function of the parameter Θ.

The next step is to compute the partial derivatives of the loss $\mathcal{J}(\Theta)$ with respect to the two different linear transformations.

$$\boldsymbol{h} = f(\boldsymbol{W_h x} + \boldsymbol{b_h})$$
$$\hat{\boldsymbol{y}} = f(\boldsymbol{W_y x} + \boldsymbol{b_y})$$

Using backpropagation we find these two partial derivatives:

$$\frac{\partial J(\Theta)}{\partial \boldsymbol{W_y}} = \frac{\partial J(\Theta)}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{W_y}}$$

$$\frac{\partial J(\Theta)}{\partial \boldsymbol{W_h}} = \frac{\partial J(\Theta)}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{W_h}}$$

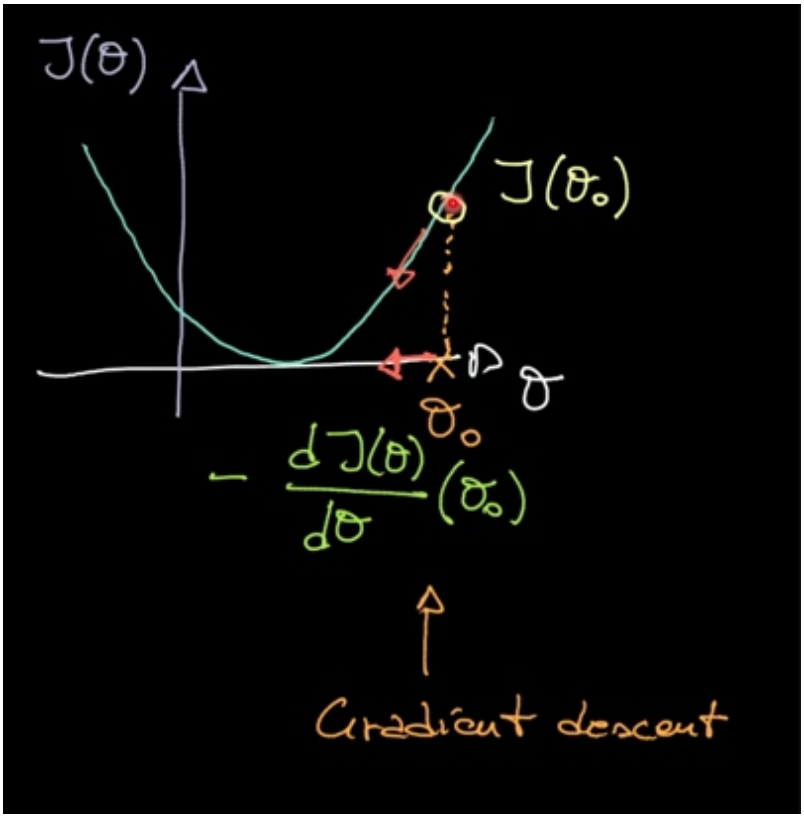


Figure 24.6: Representation: Gradient Descent

Here $J(\Theta)$ is a quadratic function, and given a point, the derivative/slope can be computed. Let Θ_0 be the starting position, the loss can be evaluate over the data set and the derivative or the slope can be figured out. The derivative of $J(\Theta)$ which is a quadratic function will be a linear function. In (figure 24.6) the slope at point Θ_0 is positive, hence the derivative is linear function which points $\nearrow$, hence, weights will be modified by stepping in the opposite direction $\swarrow$.

Chapter 25

Loss Functions in Deep Learning are non-convex

Things like regression for single layer are convex and the function you are optimizing is quadratic in x. However loss functions of pretty much every multi-layer network are non-convex and have multiple local minima. Luckily, it is shown that these local minima are all more or less equivalent. When we train a neural net with different initial conditions then the solutions we get at the end are very different to each other. However performances of all the solutions will be more or less the same.

25.1 Example: Identity Function

The figure the loss surface of a two layer neural net with one input, one hidden unit and one output and the network tries to approximate the identity function. The input and the output are both 0.5. The cost function is squared error

$$L = (0.5 - \tanh(W_1 \tanh(W_0 * 0.5)))^2 \tag{25.1}$$

The objective function looks as follows in the space of W_1, W_2. We get a saddle point at (0,0) where curvature is positive in two directions and negative in two.

Neural networks with at least one non-linear activation function in its hidden layer(s) will have non-convex loss surfaces. Convex optimization methods do not work if there are more than one global minimum. In this instance, we get two solutions symmetrical to both sides of the saddle point. Those solutions are essentially hyperbolas, because if we didn't have the hyperbolic tangents and it was completely linear then we can add one value for the weight W_0 and the inverse value for weight W_1 and get the identity function. For instance if we set W_0 to 2 and W_1 to $\frac{1}{2}$ and forget about the hyperbolic tangents then we get an identity function.

We can choose both positive and negative weights and still get a solution as we have the other solution space on the other side. This network has a very highly non-convex objective function where if we start at point that are small distance apart from each other, we can end into different

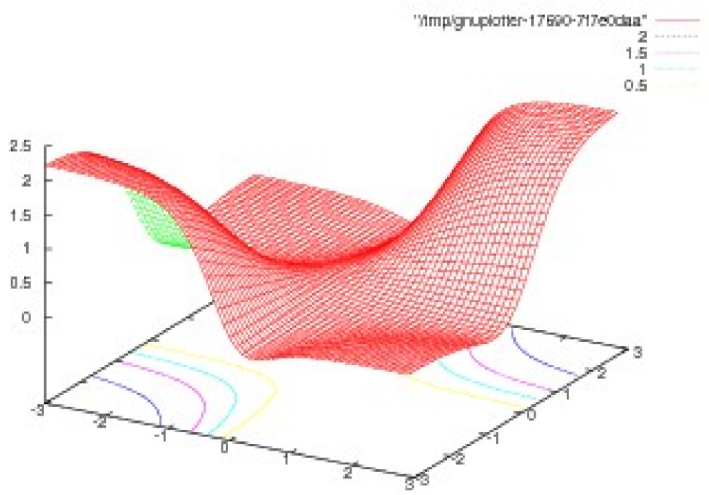


Figure 25.1: Loss Surface of Identity Function

local minima. Actually those two minima are equivalent so it does not matter where we start. There is empirical evidence that in neural networks it is very often the case, that there are lots of different solutions but they are basically equivalent.

Even if we have a simpler version without hyperbolic tangent and completely linear functions, still we see that the solution is similar. One other explanation for why regularizers are bad in the beginning of the training is that when we initialize the weights, we have to make sure the weights are initialized to non-zero values since otherwise the learning never takes off and it stays at a saddle point where there is no gradient. Similarly the backpropagation with zero weights do not update anything and the neural net never takes off. The magnitude of the weights is very important and we must pay special attention that we initialize them appropriately.

The intuition behind this is that if you have a unit with many inputs that are normalized then the weighted sum of this unit is the weighted sum of random variables with standard deviation/variance one. The variance of the output will thus be the weighted sum of the variance of the inputs. The variance of the weighted sum will be the sum of the variances of the inputs multiplied by the square of the weights. If we want our output to have variance once to preserve the variance, then we need to set the weights accordingly - to smaller values if we have many inputs and to bigger values if we have fewer inputs. Scaling the weights with a factor that is $\frac{1}{\sqrt{N}}$ would preserve variance of inputs in different layers.

Chapter 26

Convolutions

26.1 Natural Data Properties

Natural signals (e.g. audio, images, text) have 3 properties that make convolutions a good choice to process them:

1. **Stationarity**: similar patterns repeat in the data. For example, in an image, similar patches appear in different locations and not all patches are equally frequent. Formally we define this as "a process or data where the mean, variance and autocorrelation structure does not change over time." More simply put, in data, we can observe stationarity when a pattern repeats over degrees of freedom.

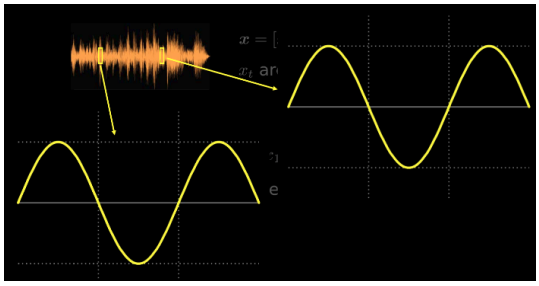


Figure 26.1: Example of stationarity - we can observe similar waveforms over different parts of one data set.

2. **Locality**: points close to one another have more information about each other than points far apart. In other words, two points far apart are less likely to have a higher correlation than two points closer to each other. For example, audio signals or pixels in an image.

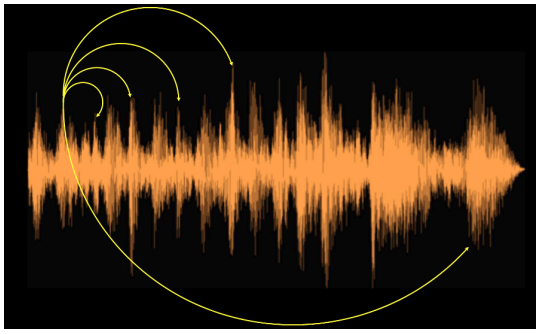


Figure 26.2: Data points are more "relevant" to other data points nearby, less relevant as you move further away.

3. **Compositionality**: the world we live in is formed from a hierarchy of structure. Each level is composed from a group of structures from the lower levels. Complex expressions are formed by a combination of its simpler constituent expressions. For example, images are composed of pixels, pixels together form edges and color patters, these together form motifs, which then form shapes, objects, scenes and so forth. This implies that a good way to identify a scene, for example, is to first understand the edges and color patterns, then the motifs and so on, which translates into the convolutional layers successfully used in this task. This compositionality characteristic of images was first introduced in biology, by analyzing how the human brain processes visual signals at each different stage of the visual cortex.

26.2 Exploiting The Properties

By making use of these properties, some simplifications can be introduced to our models:

1. **Locality** $\Rightarrow$ **Sparsity**:

   Since the information that is most relevant to identify a particular region of the signal (e.g. image) is close to that region, our models do not need to analyze regions far away from each region of interest. Thus, in an neural network, a single unit does not need to be directly connected to a large portion of the input signal. In other words, the connections can be sparse: mostly zero, except in the areas surrounding the region of interest. In biology, this phenomenon is called the Receptive Field: "an individual neuron relates to a specific sensory space (e.g., the body surface, or the visual field) in which a stimulus will modify the firing of that particular neuron"; aka, neurons and their analogous receptive fields are highly localized.

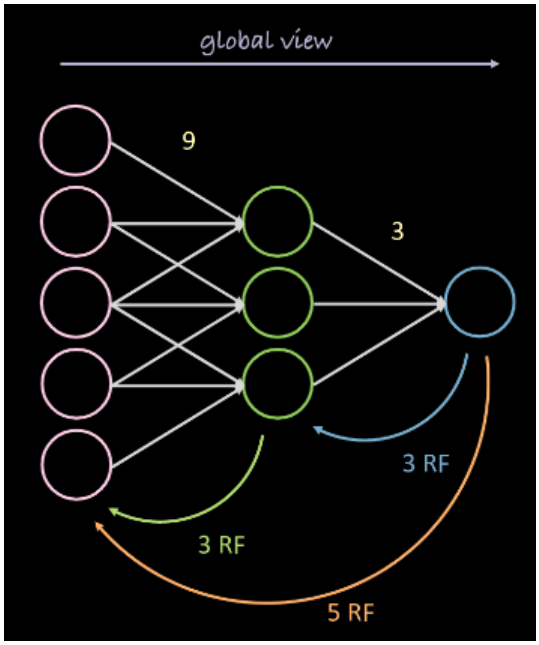


Figure 26.3: Like receptive fields, we can restrict input connections from incoming layers, instead of being fully connected.

2. **Stationarity ⇒ Parameter Sharing**:

   Since we know only a portion of the input needs to perfuse to each unit, we then need to determine which parameters to connect the adjacent layers. We have previously explained that similar patterns repeat over and over in the data, so it becomes clear that sharing parameters across the input space is a good practice. We can have multiple sets of parameters (i.e. kernels), each that focus on identifying a specific pattern, and use each of these sets across the whole input data.

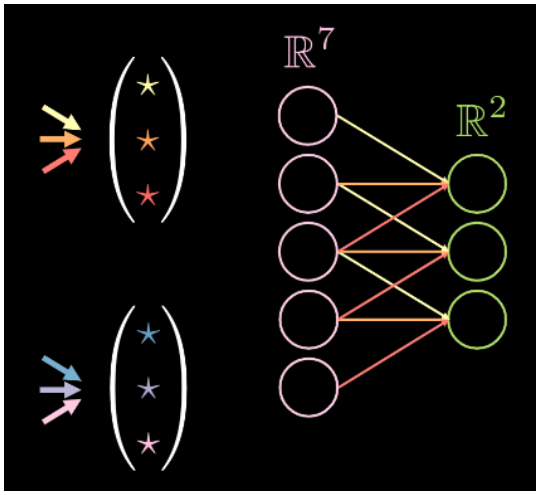


Figure 26.4: Example of parameter sharing using kernels. Colored connections refer to kernel space

26.3 Resulting Improvements

The use of Sparsity and Parameter Sharing leads to:

1. **Faster convergence** → Fewer weights to tune and ability to optimize the same parameters using multiple parts of the network/data.
2. **Better model generalization**→ Fewer parameters leads to less overfitting.
3. **Models not constrained to input size**→Can keep applying same sets of parameters to small regions of the input independent to its size.
4. **Kernel independence**→ leads to higher parallelization capabilities.
5. **Reduced amount of computation**→ efficiency!

26.4 Notes

Two important aspects to keep in mind:

- **Kernel Format**: In PyTorch, the order that the kernels are stored in the tensor is:

$$\underset{\#kernels}{\mathrm{N}} \times \underset{\#channels}{\mathrm{C}} \times \underset{h \times w \text{ of kernel}}{K} \tag{26.1}$$

- **Zero-Padding**: Operation of introducing zeros to the borders of the input. It is commonly used in order to maintain the size of the input in the output after a convolutional transformation.

Chapter 27

Visualizing 2D interpolation

We take the data simulated in the 04-spiral_classification notebook ("Spiral Classification"), consisting of 3,000 samples of dimension 2 (Fig. 27.1). The data is generated from the following expression:

$$X_c(t) = t \begin{bmatrix} \sin \frac{2\pi}{C}(2t + c + 1) + \mathcal{N}(0, \sigma^2) \\ \cos \frac{2\pi}{C}(2t + c + 1) + \mathcal{N}(0, \sigma^2) \end{bmatrix}$$

where $0 \leq t \leq 1$ and classes $c = 1, ..., C$.

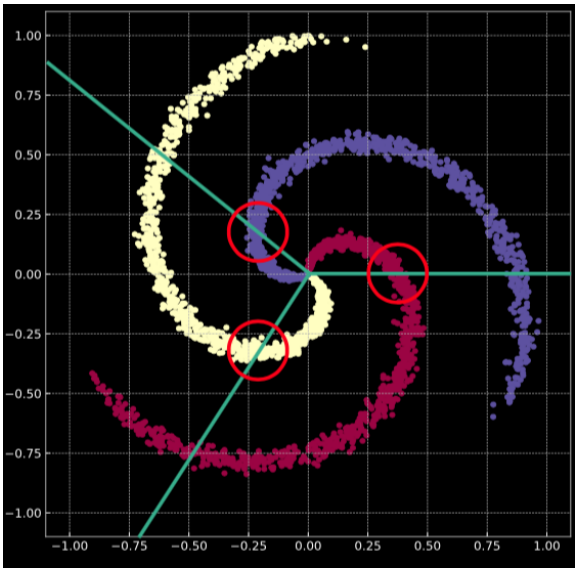


Figure 27.1: 3 non linearly separable curves consisting in 3,000 samples from $X \in \mathbb{R}^2$.

To classify the data into the three categories, we are going to use a three layer net with:

(1) 2 input units (dimensionality of the data).

(2) A 100-units hidden layer (used to increase the space and extract features).

(3) Followed by a non-linear ReLU activation function.

(4) We then go back down to 2D again which is useful for visualization.

(5) Lastly we can bring it back up to 3D with a final linear transformation to a 3-unit output layer with softmax activation function for classification (Fig. 27.2). The last linear transformation involves the multiplication of $\boldsymbol{A^{(2)}}$ and $\boldsymbol{W^{(2)}}$, where the different rows of $\boldsymbol{W^{(2)}}$ are the 2D vectors that will point at the different classes in the 2D space. In this representation, the different classes are linearly separable.

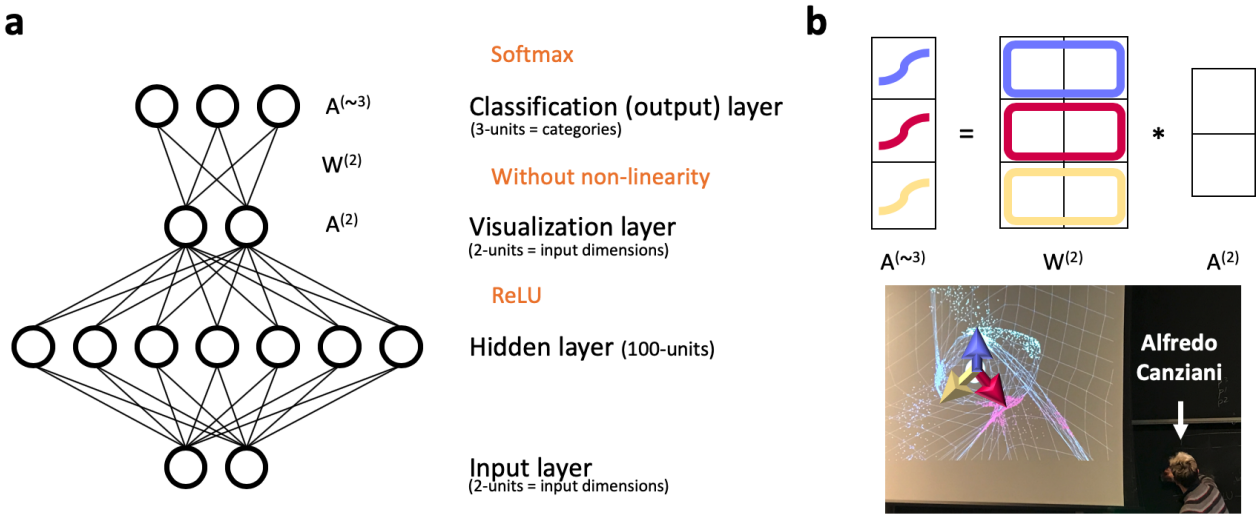


Figure 27.2: Neural network architecture for classification and visualization in the input space. (a) Architecture of the neural network, which takes 2D inputs and visualizes in the same dimensional space before classification. (b) Linear transformation that takes $\boldsymbol{A^{(2)}}$ and returns $\boldsymbol{A^{(3)}}$ via multiplication with $\boldsymbol{W^{(2)}}$. Rectangular color boxes in $\boldsymbol{W^{(2)}}$ represent the three 2D vectors that will define each class. Calculation of $\boldsymbol{A^{(3)}}$ is followed by a softmax activation function for classification. In the bottom half of (b), the transformed data after and the population vectors (plot credit: Alfredo Canziani).

In order to visualize each transformation, we do a linear interpolation from the input to the output at the 2D hidden layer using:

$$(1-\alpha)(x^{(1)}) + \alpha\phi(x^{(i)}) \quad \text{where } 0 \leq \alpha \leq 1$$

Chapter 28

Convolution Demonstration

28.1 Natural Signal Patterns

Neural networks can be used to model audio, image, text, or other signals. The signals are represented as sequences of scalars. Audio is often represented as waveform heights, images are often represented as pixel values, and text is often represented as one hot vectors.

Natural signals (not artificial or synthetic) tend to exhibit two important patterns, which are both crucial for convolutional neural networks.

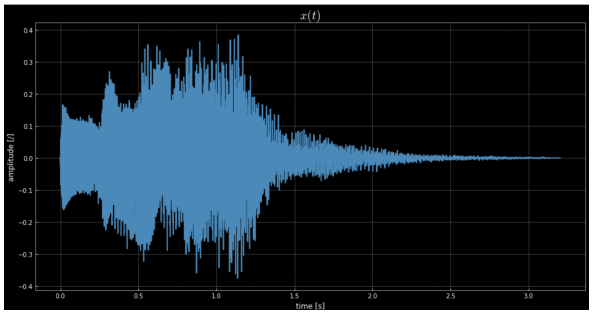
1. Stationarity - The waveform heights of audio signals form a sinusoidal pattern with similar sub-segments of peaks and valleys occurring repeatedly throughout the signal.

2. Locality - The correlation is high between two peaks in the waveform at nearby points in time, but the correlation is low between distant peaks. Sounds have "local" properties such as being more transient or smoother in the time-frequency domain. If an audio signal were shuffled, so that the index of the data no longer represented a position in time, the audio signal would no longer adhere to the stationarity and locality assumptions.

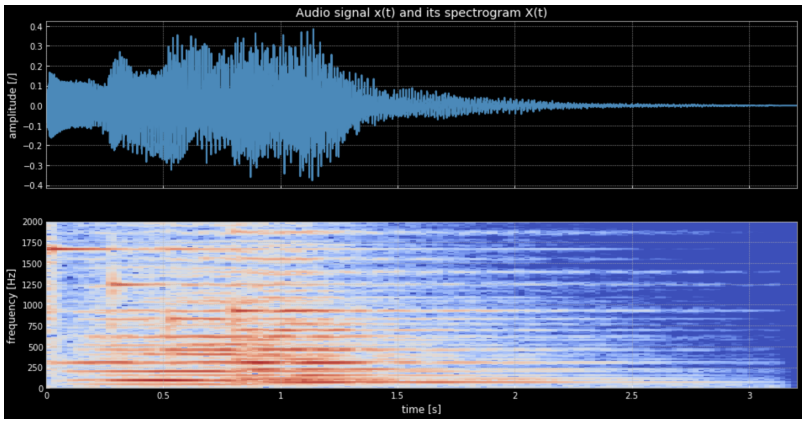
28.2 Audio Example

Here, we use a python library "librosa", which is for audio and music analysis. Select natural signal:

`audio.wav`

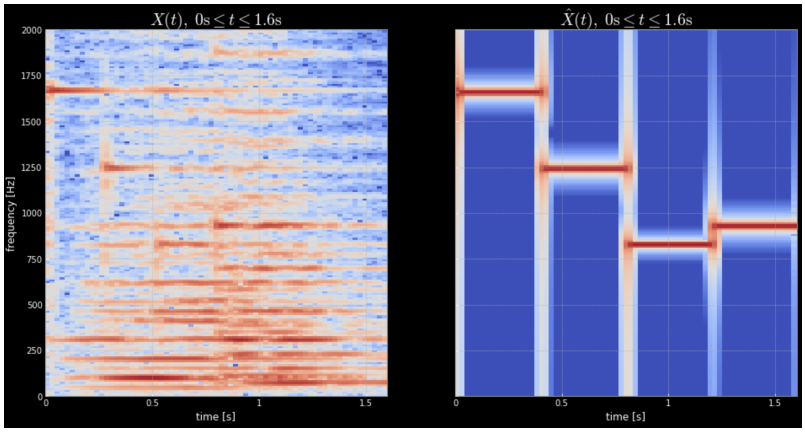


The wave form constitutes of the amplitude in the time domain. To analyze it in the frequency domain we use discrete fourier transform. As we want to know the frequencies over time we do fourier transform over windowed signal. The resulting graph is called a spectrogram, which represents the signal in the frequency domain.



The challenge is to figure out which corresponding piano keys we should play to reproduce the audio. We can either pick the frequencies from the spectrogram or we can use all frequencies of piano keys. When we do a convolution of the audio sample with this reconstructed sample we will observe that the frequencies of audio sample separated.

In the lab we did convolution with reconstructed sample which consisted of the frequencies we guessed from spectrogram. The spectrogram of the natural signal and the one we generated can be seen below.



Chapter 29

Automatic Differentiation

Automatic differentiation is a hybrid of symbolic differentiation and numerical differentiation. It is extremely efficient when it is desired to differentiate functions of the form

$$y_n = f_n(w_{n-1}, f_{n-1}(w_{n-2}, \cdots (f_0(w_0, x_0))$$

When back-propagation is performed, there is a desire that $\partial y_k / \partial w_\ell$ can be explicitly expressed in terms of y_k. Thankfully, there are packages that handle auto-differentiation such that it does not have to be done manually.

In `pytorch`, each tensor has an attribute called `grad_fn` which handles the process of calculating $\partial y / \partial x$ with a given y. Note that `grad_fn=None` for those tensors that cannot be differentiated. For each derived tensor, `pytorch` internally builds a computation graph. Moreover, gradients can be constructed implicitly with the `backward()` method.

For both memory and efficiency reasons, the computation graph would be discarded once it is backwarded. By default, a computation graph can not be backwarded twice. Once the parameters are updated, the backwarded scalar must be recomputed before the computation graph can be constructed at a new position.

In general the `backward()` method requires an input of the same size as the backwarded tensor.

```
y.backward(h)
# h * J(x)  = x.grad, where J(x) is the Jacobian calculated from y
# though pytorch does not calculate and store Jacobian internally
```

By default, explicitly constructed tensor `x` has `x.require_grad=False`. The `require_grad` attribute can be manually turned on or off for leaf nodes on the computation graph (they do not depend on another tensor). If it is desired to turn off a `requires_grad` for an intermediate result, it must be copied and without reference. The `.detach()` method is handy for this purpose.

```
n = 3
x = torch.randn(n, requires_grad=True)
w = torch.ones(n, requires_grad=True)

z = w @ x

z1 = z.detach()
```

```
z2 = w * z1 @ x;
z2.backward()
print(w.grad, z1.grad, x.grad, sep='\n')
```

The output would be:

```
tensor([-0.2010, -0.0613,  0.5340])
None
tensor([0.5213, 0.5213, 0.5213])
```

Chapter 30

Random Projections

The dimensionality of data significantly impacts how well a machine learning model can generalize. Consider unit orthogonal vectors in space. With increasing dimensions, the number of orthogonal vectors increase exponentially (source). To generalize well, the model needs at least `exp(d)` points, where `d` is the data dimension. This is because for a new data point, the models should find another point in the dataset which is reasonably close to it.

The Jupyter notebook illustrates orthogonality of random projections using a simple example. It plots the projection `p` of a random matrix `A` with one of its randomly chosen rows `a`. Since `A` is a random matrix of unit vectors (normalized with unitary L-2 norm), `p` will have a value of `1` for one row.

Figure 2 shows the matrix plots for `d=3`. There are at least three rows in the projections which have a value greater than `0.800`. Figure 3 shows the matrix plots for `d=5`. There are no rows with a value close to `1`, except for the one corresponding to `a`.

The intuition is that in low dimensions random vectors point roughly in same directions. However in high dimensions, they point orthogonal to each other. As the dimensionality of the data is increased, more points becomes further apart and there are less matches.

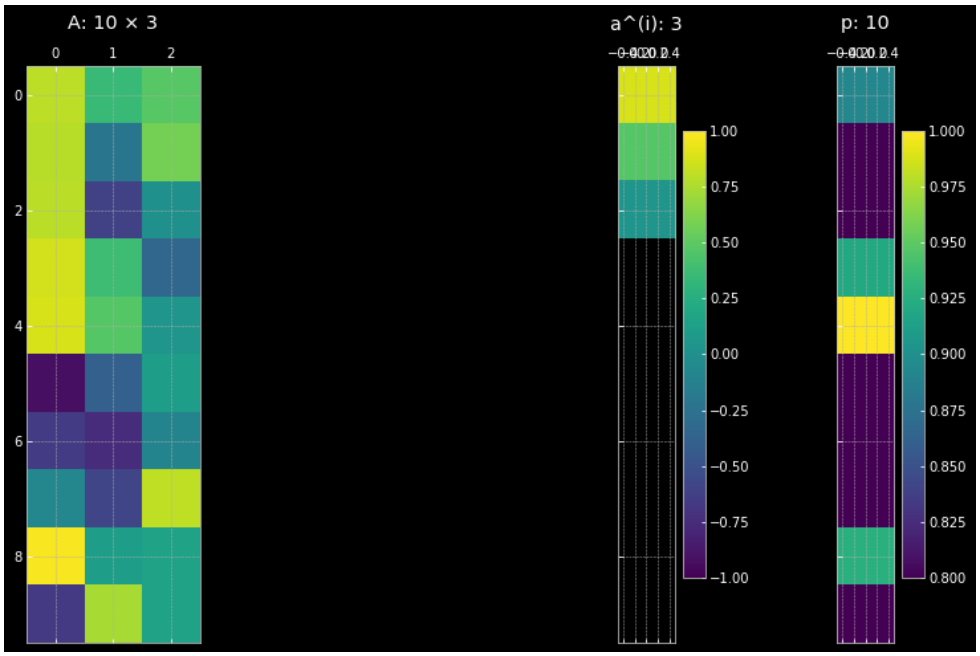


Figure 30.1: Projecting p for A with dimensions (10, 3)

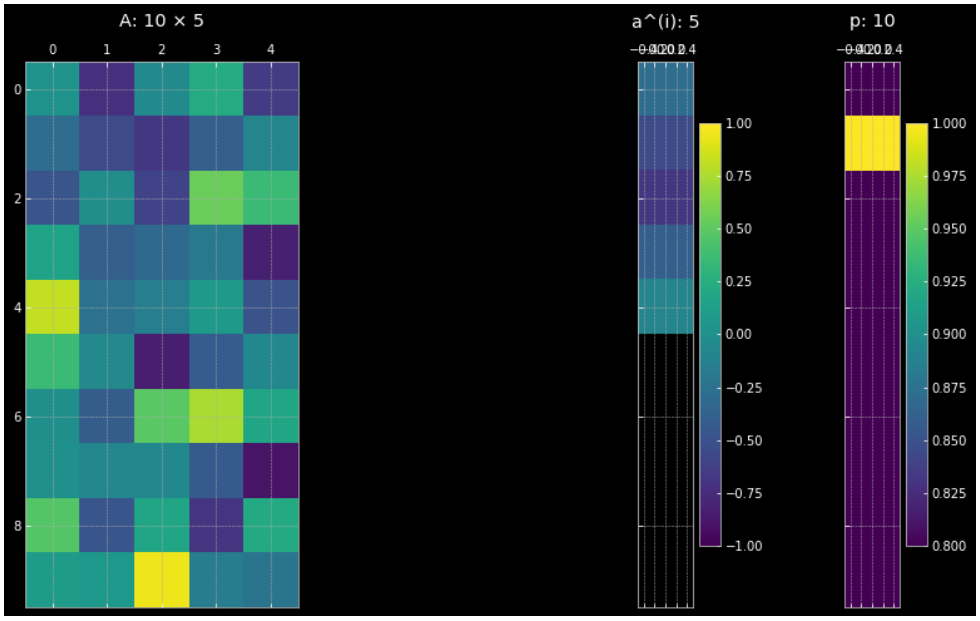


Figure 30.2: Projecting p for A with dimensions (10, 5)

Chapter 31

Comparison among Different Image Classification Neural Nets

The age of information explosion is characterized by an overflow of data in all forms. Images, in particular, are among one of the most common types of unstructured data. Nowadays, deep neural networks prove to be an excellent model for making sense of images. One crucial objective of these models is to classify the object that appears in an image. Nevertheless, training a deep neural network is an arduous task. It entails the following:

1. **Computational Power.** Training a neural network is a computationally expensive process. It requires a powerful CPU (or GPU) and generally takes a substantial amount of time. The actual training time hinges on the complexity of the model, the number of training instances, etc.

2. **Amount of Data.** In addition to computation, sufficient amount of training data is also a must. To obtain a neural network that is able to make accurate predictions, we need an immense set of labeled data.

By far, the most popular dataset of pre-labeled images in the domain of deep learning is the *ImageNet*. *ImageNet* is a public dataset that consists of 1,000 classes, each of which has 1,000 images. Remarkably, it serves as a common metric for the performance of image classification models, as we will illustrate in the following sections.

31.1 Challenges in ImageNet Classification

In the ImageNet classification challenge, the ultimate goal is to obtain the highest accuracy in the classification problem framework. However, in practical applications, the computational requirements and resource utilization are essential for the implementations and deployments of DNN models. Comparisons between these models are made from the following perspectives: **accuracy, memory footprint, parameters, operations count, inference time and power consumption**.

31.2 Brief Summary of Deep Neural Nets Models

Let's introduce a few Deep Neural Nets Models commonly used for ImageNet classification: AlexNet, GoogLeNet, VGG, ResNet, and Inception.

- **AlexNet:** Winner in the 2012 ILSVRC, the first entry that used a Deep Neural Network. Alexnet contains 5 convolutional layers and 3 fully-connected layers. Techniques used in the model are still being used today, such as data augmentation and dropout.
- **GoogLeNet:** The winner of ILSVLC 2014. GoogLeNet designs a Inception Module, which concatenates the output of multiple convolution filters or pooling together. This allows the model to take advantage of multi-level feature extraction, global and local at the same time. In terms of practical applications, GoogLeNet has better parameter utilization. As a fun fact, Inception Networks are named after the Inception movie meme that says "We need to go Deeper".
- **VGG:** The winner in ILSVRC 2014 developed by Karen Simonyan and Andrew Zisserman. VGG achieves high accuracy by pushing the depth of DNN to 16-19 weight layers. By far, VGG is the most expensive architecture both in terms of computational requirements and number of parameters.
- **ResNet:** Developed by Microsoft. It was the first time we were able to train a network with more than 100 layers. 20 layers were the most we could train back than before the appearance of the ResNet due to the vanishing gradient problem. Vanishing gradient is a phenomenon that appears during backprop and causes the parameters in the earlier layers not able to learn from gradient descent.
- **Inception:** A revenge of Google since GoogLeNet was beaten by Microsoft's ResNet. It strives to use the least amount of the computation power to achieve a lightweight network.

Some images, though only labeled as only certain object, might contain several objects in the image. Neural networks might produce an output that is correctly contained in the image, but not labeled. For example, an image might have a dog and cherries, but it is labeled as dog, and the networks might classify the image as cherries instead of a dog. To account for this issue, people include the top 5 metrics to be scored as correct answer.

31.3 Resource utilization of different DNN architectures

It is discovered that the resource utilization of AlexNet is strongly associated with the choice of batch size during the training. In this section, we are going to mainly focus on this surprising finding.

In the following graphs, we observe that the use of energy, memory, inference time remains the same for most of the Deep Neural Nets when we change the batch sizes. Nevertheless, for AlexNet, the inference time per image is less, and the net power consumption is more, when batch size is increased. The speedup is due to a weak optimization of AlexNet's fully connected layers with small batch size, which causes the poor system memory and power utilization.

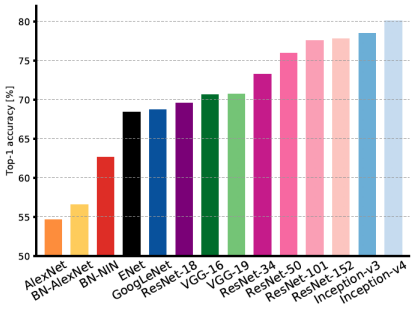


Figure 1: **Top1 *vs.* network.** Single-crop top-1 validation accuracies for top scoring single-model architectures. We introduce with this chart our choice of colour scheme, which will be used throughout this publication to distinguish effectively different architectures and their correspondent authors. Notice that networks of the same group share the same hue, for example ResNet are all variations of pink.

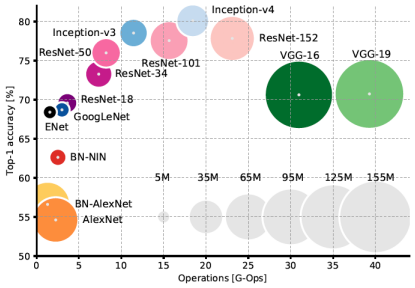


Figure 2: **Top1 *vs.* operations, size $\propto$ parameters.** Top-1 one-crop accuracy versus amount of operations required for a single forward pass. The size of the blobs is proportional to the number of network parameters; a legend is reported in the bottom right corner, spanning from 5×10^6 to 155×10^6 params. Both these figures share the same y-axis, and the grey dots highlight the centre of the blobs.

Figure 31.1

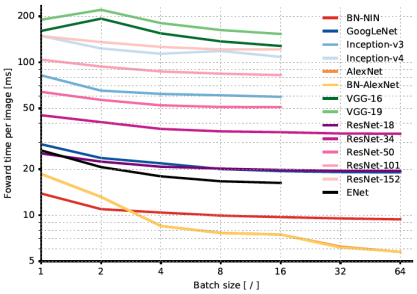


Figure 3: **Inference time *vs.* batch size.** This chart show inference time across different batch sizes with a logarithmic ordinate and logarithmic abscissa. Missing data points are due to lack of enough system memory required to process larger batches. A speed up of $3\times$ is achieved by AlexNet due to better optimisation of its fully connected layers for larger batches.

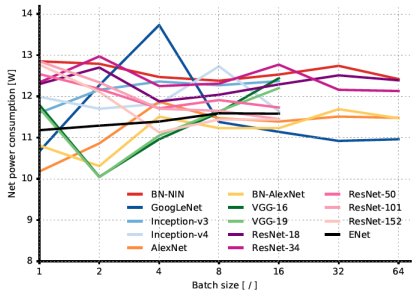


Figure 4: **Power *vs.* batch size.** Net power consumption (due only to the forward processing of several DNNs) for different batch sizes. The idle power of the TX1 board, with no HDMI screen connected, was $1.30\,\mathrm{W}$ on average. The max frequency component of power supply current was $1.4\,\mathrm{kHz}$, corresponding to a Nyquist sampling frequency of $2.8\,\mathrm{kHz}$.

Figure 31.2

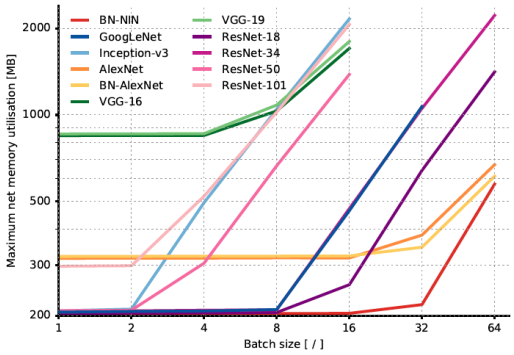


Figure 5: **Memory *vs.* batch size.** Maximum system memory utilisation for batches of different sizes. Memory usage shows a knee graph, due to the network model memory static allocation and the variable memory used by batch size.

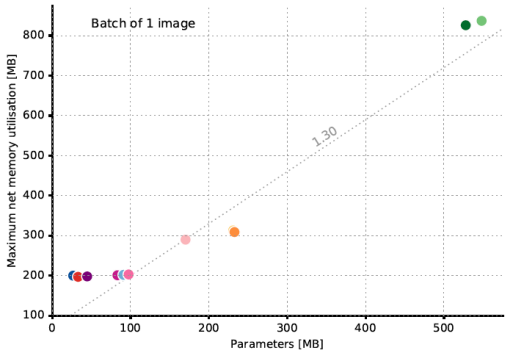


Figure 6: **Memory *vs.* parameters count.** Detailed view on static parameters allocation and corresponding memory utilisation. Minimum memory of 200 MB, linear afterwards with slope 1.30.

Figure 31.3

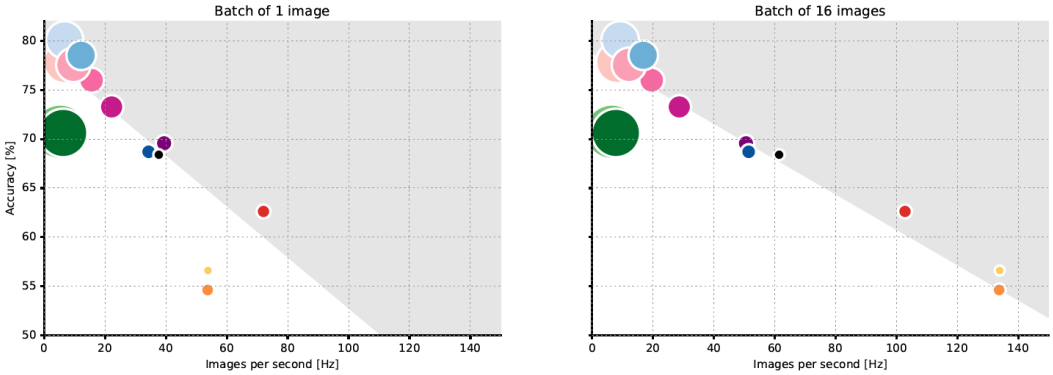


Figure 9: **Accuracy *vs.* inferences per second, size $\propto$ operations.** Non trivial linear upper bound is shown in these scatter plots, illustrating the relationship between prediction accuracy and throughput of all examined architectures. These are the first charts in which the area of the blobs is proportional to the amount of operations, instead of the parameters count. We can notice that larger blobs are concentrated on the left side of the charts, in correspondence of low throughput, *i.e.* longer inference times. Most of the architectures lay on the linear interface between the grey and white areas. If a network falls in the shaded area, it means it achieves exceptional accuracy or inference speed. The white area indicates a suboptimal region. *E.g.* both AlexNet architectures improve processing speed as larger batches are adopted, gaining 80 Hz.

Figure 31.4

- **Accuracy:** A linear fit of the accuracy shows all architecture trade accuracy vs. speed in figure 31.4. Since power consumption is constant, we can presume that there exists an theoretical upper bound in accuracy when there is a energy constraint.
- **Inference time:** There is a 3 fold improvement for BN-alexNet when batch size is increased, as shown in figure 31.2.
- **Power Consumption:** Lower power consumption is associated with slower forward times per image. Low power consumption networks are preferred for mobile devices, and thus people start to pay more attention on this matter as more and more mobile devices are implemented with neural networks.
- **Memory:** In figure 31.3, the initial allocation of the memory is never less than 200 MB. Up to a certain point, the memory required starts to grow proportionally with the number of images/batches. Memory becomes a crucial constrain when we want to employ neural networks in our mobile devices. In addition, to calculate the size of the networks, we multiply the number of parameters by 4. For example, the size of GoogLeNet is 20 MB (5 million parameters time 4).
- **Operations:** There is a linear relationship between operations count and inference time per image.
- **Parameters:** Parameters utilization is closely related to information density in Deep Neural Network. It is also inefficient to not use the fully learning power when training the parameters. A more compact architecture is often wanted in newer design of Deep Neural Nets.

31.4 Residual(skip) Connections

Now we introduce the concept of **residual connections**.

In general, the more hidden layers a neural network has, the more accurate prediction it is able to yield. As a result, people often favor a deeper neural network as opposed to a shallower one. Nevertheless, problems may arise when the neural network attempts to learn a simple function, where deepness often leads to lower accuracy. This is known as the *degradation problem*.

One popular solution is to skip a few hidden layers at training time, as shown in the figure above. In the residual net, the residual layer tries to learn from residual $R(x)$ rather than the true output $H(x)$. Hence, it is called the *residual block*.

In other words, we provide an alternate route to some of the hidden layers – an identity function. When some layers proved to be counterproductive, the identity function of previous layer will take over and pass to next layer. By doing so, we can achieve a faster computation time and generate a more smoothie loss surface.

The two graphs illustrate the loss surfaces obtained with and without skipping connections. Note that, by skipping a number of hidden layers, we end up with a much smoother loss surface compared to the one resulted from the absence of skip connections. Therefore, skip connection is an extremely useful practice in training a deep neural network.

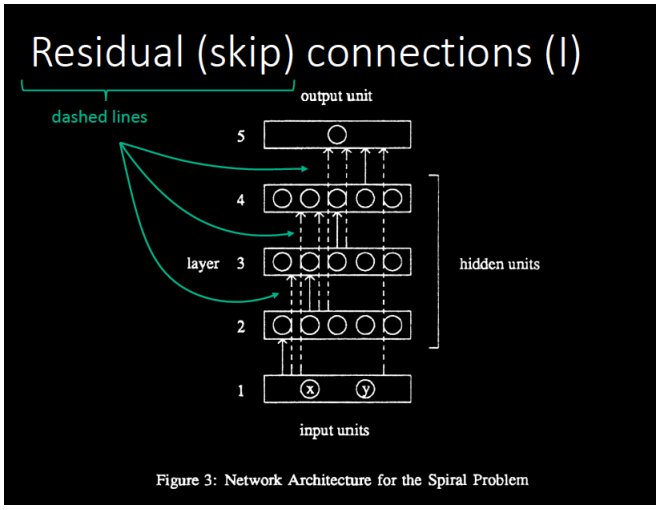


Figure 3: Network Architecture for the Spiral Problem

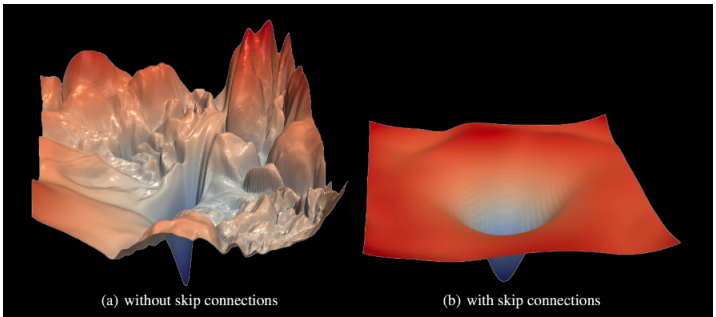


Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The vertical axis is logarithmic to show dynamic range. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

31.5 Summary

A neural network heavily relies on computational power and data. Neural networks are often used to classify images, yet we inevitably face a trade-off between accuracy and other practical considerations such as memory. A batch size sufficiently large is required to optimize the performance of a neural network. To smooth a loss surface, we often conduct skip connections.

Chapter 32

Long term vs short term memory

32.1 Why Is RNN Useful, And What Are The Drawbacks?

In lab8 we saw that RNN is useful to learn dependencies over time, and hence can model time dependent sequences effectively. This means RNN can store context/history at any given time t, which can be used along with the input to predict the output. More details about RNN can be found in Section 13.1.

However, practically we observe that as the sequence length grows over time, RNN is not able to learn long term dependencies. This is because of the **vanishing/exploding gradient** during **back-propagation through time**. Back-propagation through time is the method of calculating gradients by unfolding the network in time and then applying normal back-propagation to the unfolded network. If the the length of input sequence is large, then network is unfolded many number of times. The consequence of this is that during back-propagation, gradient is multiplied many times (more the sequence length, more the number of multiplications).

This causes the, gradient to either explode or vanish, which is undesirable. To solve the issue of exploding gradient, we generally "clip" the gradient to avoid growing it bigger than a threshold value. However RNN performs poorly for large length sequences because of vanishing gradient problem.

32.2 Why Is LSTM Useful?

Vanishing gradient problem can be solved efficiently by using LSTMs. LSTM stands for Long Short term Memory and it has been proven effective to model long term dependencies more effectively than RNN, meaning that LSTM has greater memory than RNN. The reason for this is that LSTMs maintain a cell state which runs straight down the entire chain, with only some minor linear interactions. They have the ability to either add or remove information to this cell state using gates namely the forget, input and output gates. More details about LSTM can be found here

The reason why vanishing gradient is handled better in case of LSTM is that gradient flows back effectively through the network because of the presence of relevant gates which makes it very easy for information to just flow along the cell state unchanged.

32.3 Lab 8 - Result Analysis

In lab8, we try to classify a sequence of characters into categories Q,R,S,U. This classification is done on the basis of two characters X,Y which can be present anywhere in the sequence. In this experiment we compare the ability of RNN and LSTM to learn dependencies for different sequence lengths. The Difficulty and configurations of the experiment is explained in detail in section 40.1. Here is the table of the test accuracies -

DIFFICULTY	**RNN (10 epochs)**	**LSTM (10 epochs)**	**RNN (100 epochs)**	**LSTM (100 epochs)**
Easy	60.58%	79.33%	87.30%	87.50%
Normal	24%	26%	71%	85%
Moderate	24.60%	28.43%	26.65%	84.37%
Hard	23.19%	25.10%	24.70%	25.30%

As we can see that LSTM outperforms RNN in almost all the cases. We also observe that RNN and LSTM are not able to learn the sequence when ran for 10 epochs in case of MODERATE, NORMAL and HARD LEVEL. When ran for 100 epochs, RNN and LSTM are not able to learn for HARD level, whereas they are able to learn other configurations.

Chapter 33

Regularization

33.1 Problem

We attempt to study the effect of L1 regularization,L2 regularization and Dropout in sentiment analysis of IMDB dataset. We split our data into training (17500 samples), valiation (7500 samples) and test set(2500 samples).

33.2 Data processing

We follow the following steps while processing the IMDB dataset

1) Tokenization: break sentence into individual words
   - Before: "PyTorch seems really easy to use!"
   - After: ["PyTorch", "seems", "really", "easy", "to", "use", "!"]
2) Building vocabulary: build an index of words associated with unique numbers
   - Before: ["PyTorch", "seems", "really", "easy", "to", "use", "!"]
   - After: "Pytorch: 0, "seems": 1, "really": 2, ...

   We use a vocabulary of 1000 words in this example
3) Convert to numerals: map words to unique numbers (indices)
   - Before: "Pytorch: 0, "seems": 1, "really": 2, ...
   - After: [0, 1, 2, ...]
4) Embedding look-up: map sentences (indices now) to fixed matrices

[0.1, 0.4, 0.3 , [0.8, 0.1, 0.5], ...]

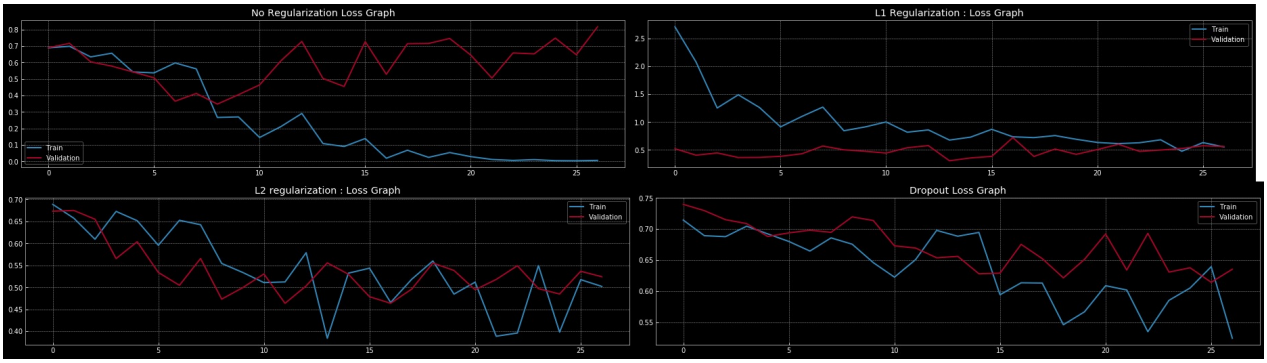


Figure 33.1: Loss Graph

33.3 Architecture

We use a Feed-forward neural network(FNN) for our task. (Not RNN/LSTM/GRU). FNN does not contain any cycles or loops in the network. There are no feedback connections in which outputs of the model are fed back. It cannot handle flexible sequence lengths so we have to fix the length of the input. We use a 3 layer FNN where the first layer is the embedding layer. This is followed by a hidden layer with RelU activation. We have a third output layer. We apply a sigmoid function to our output so that our loss doesn't have to go through sigmoid again. There are two sentiments for a review - negative or positive. This is a binary classification problem so Binary Cross Entropy loss (BCE loss) is used.

33.4 Experiments

We experiment with the following regularizations in the classification problem:

- No regularization
- Add L1 regularization
- Add L2 regularization
- Add Dropout

When no regularization is applied, the network overfits the training data and the validation loss is much larger than the training loss. When we apply L1, L2 or Dropout regularization, the validation loss is comparable to the training loss and the network does not overfit anymore.

We see that when we apply dropout or no regularization to the network, the weights are more distributed. For L1 and L2 regularization though, most of the weights are close to 0. The L2 weights distribution is more spread out than L1 as it is like a Gaussian distribution, while L1 weights are like Laplacian distribution.

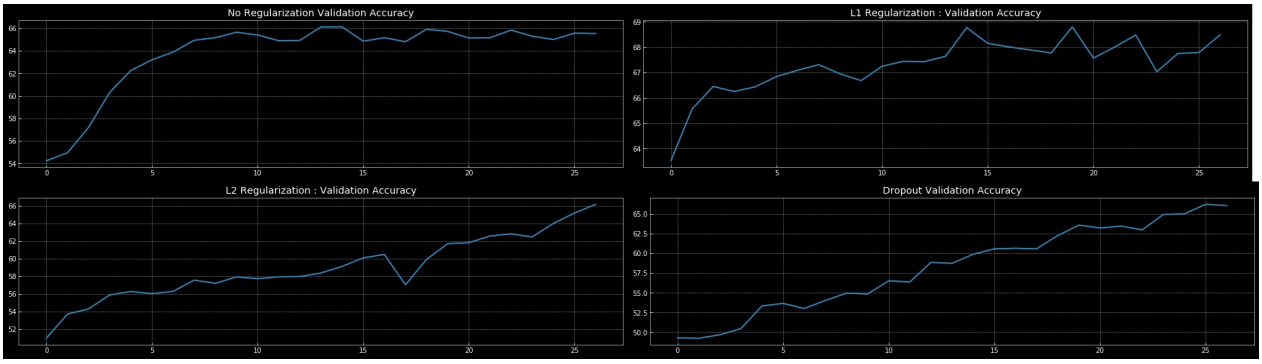


Figure 33.2: Validation accuracy

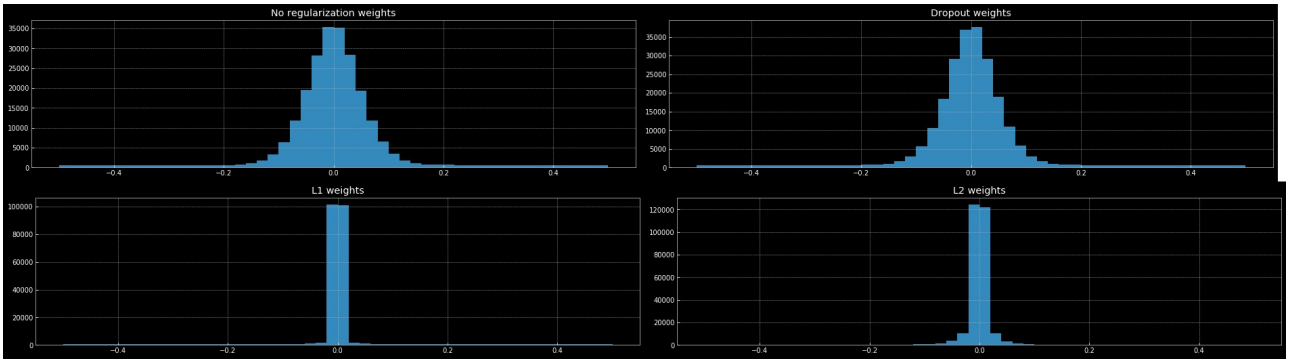


Figure 33.3: Weights distribution for different regularizations

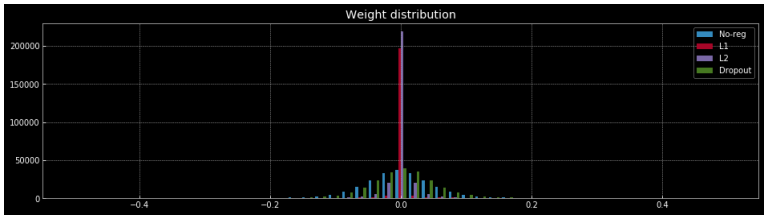


Figure 33.4: Weights distribution comparison

Chapter 34

Variational Auto-encoders and Generative Adversarial Networks

34.1 Variational Auto-Encoders

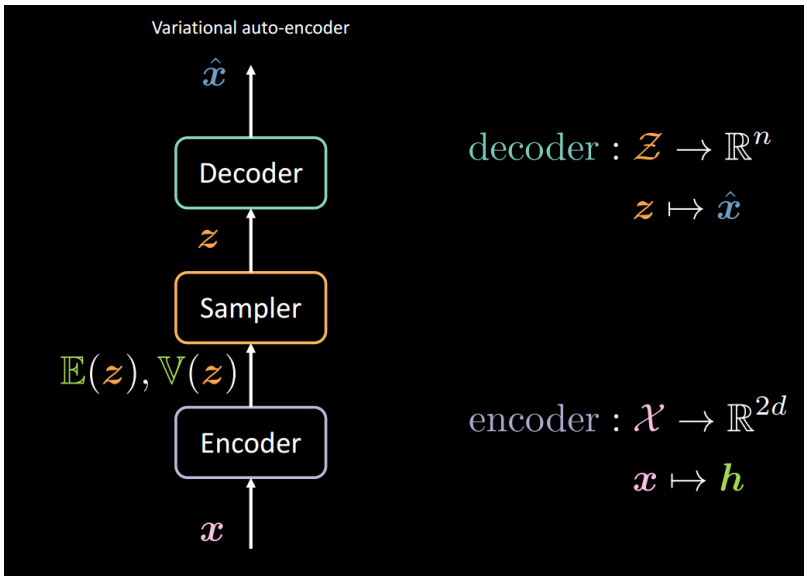


Figure 34.1: Architecture of Variational Autoencoder

There are two types of Auto-Encoders: Over-complete and Under-complete. A type of constraint that can be applied which is information bottleneck is reduction in the size of intermediate layer which leads to under-complete auto-encoder. In over-complete auto-encoder there is larger intermediate layer so we can extract better meaningful features. Variational autoencoders (VAE) inherit the architecture of the traditional autoencoders – they also consist of two main components: encoder

and decoder. However, there is a key difference between them – while encoder in vanilla AE outputs a hidden representation h of the input x, VAE encoder produces two vectors – expectation $E(z)$ and variance $V(z)$ over latent variable z (Figure 34.1). This property allows us to produce stochastic representations of the input. This is achieved by sampling a vector from the standard normal distribution with zero mean and unit variance and shifting and rescaling vector elements according to estimated $E(z)$ and $V(z)$ respectively:

$$\boldsymbol{z}' = \mathbb{E}(\boldsymbol{z}) + \boldsymbol{\epsilon} \odot \sqrt{\mathbb{V}(\boldsymbol{z})}$$
$$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathcal{I}_d)$$

After that, sampled representation z' is decoded into output $\hat{x}$. In order to make sure that this output is close to the input x, we introduce a reconstruction component of the loss function – $\ell(x, \hat{x})$ (e.g., MSE loss or binary cross-entropy depending on the nature of the data).

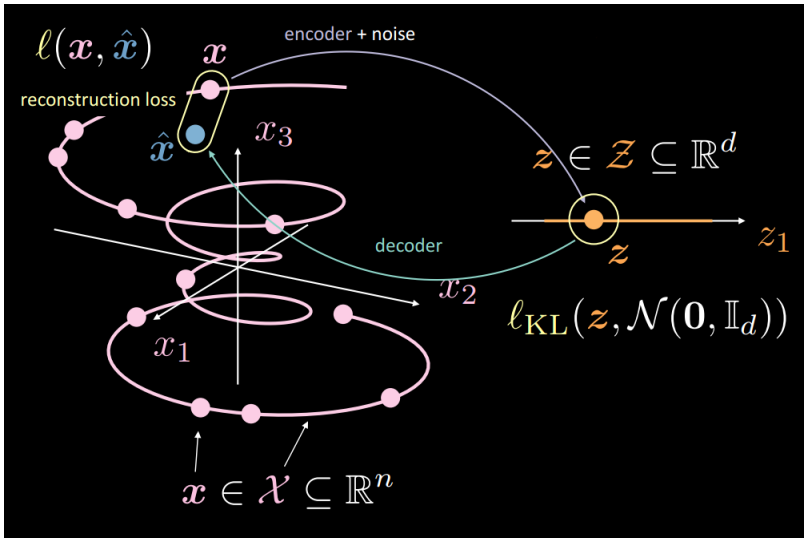


Figure 34.2: Visualization of encoding and decoding processes

However, as there is no limit on the values of $E(z)$ and $V(z)$, the encoder can learn to push different training samples very far apart in the latent space (Figure 34.3a), effectively reconstructing the training data. In order to avoid this, we introduce a constraint by adding KL-divergence component to the loss function.

$$\ell(x, \hat{x}) = \ell_{rec} + \beta \ell_{KL}(z, \mathcal{N}(0, \mathcal{I}_d)) \simeq \ell_{rec} + \beta \sum_{i=1}^{d} (\mathbb{V}(\boldsymbol{z}) - \log \mathbb{V}(\boldsymbol{z}) - 1 + \mathbb{E}(\boldsymbol{z})^2)_i$$

Intuitively, this new term forces learned representations of the training samples to have unit variance and to be as close to zero as possible (Figure 34.3b). β is a hyperparameter that controls the tightness of the constraint. To sum up, reconstruction loss ensures that the different "bubbles" do not overlap, so we can still reconstruct the different inputs, while KL-divergence loss confines them to the most compact space possible. This feature of VAE leads to a smooth learned latent

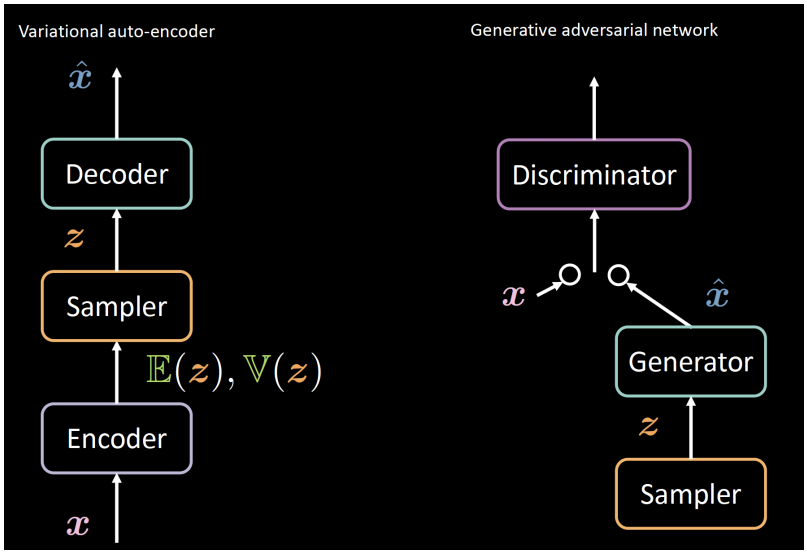
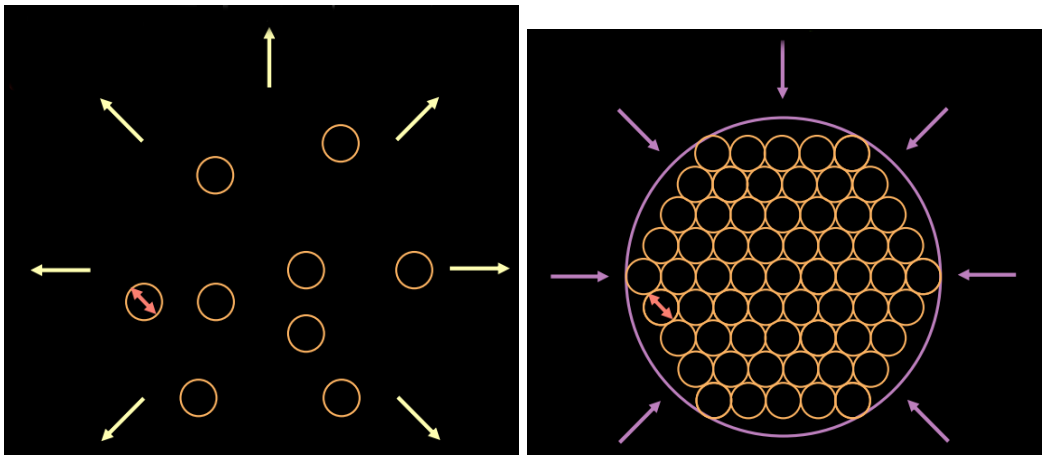


Figure 34.4: Architecture of GAN

spaces, allowing interpolation between the different samples and classes.



(a) Samples get pulled apart by reconstruction loss

(b) Samples get pushed towards zero by KL-divergence loss

Figure 34.3: Effects of the different loss components

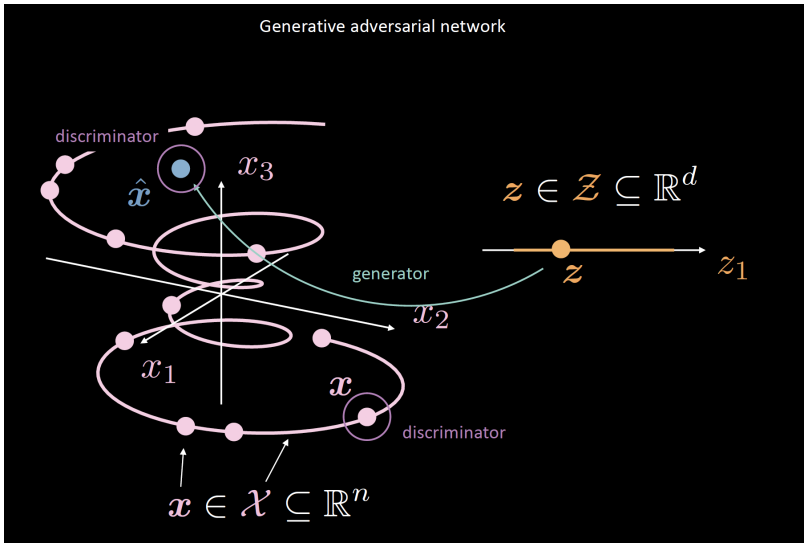


Figure 34.5: Visualization of encoding and decoding process

34.2 Generative Adversarial Network

34.2.1 Introduction

As we can see in the figure. The GAN model is similar to the VAE model.

Consider a guy who makes fake money. The $\hat{x}$ produced by sampler+generator can be thought as the fake money he makes. Then he sends the money to someone(the discriminator) who wants to tell apart fake money from the genuine one (x). If the fake money is not 'good' enough, the discriminator will sends some feedback(sent through backpropagation) and the guy can improve his skills(gradient descent), until the discriminator can not tell apart the fake one from the genuine one.

From the sampler we come up with some noise. On top of the sampler we have the generator which tries to create authentic data, similar to the decoder to VAE, both of which are generative networks. From the generator we get our estimate $\hat{x}$on the manifold. The estimate, together with our real x, is sent to a discriminator which tries to tell apart x from $\hat{x}$ that is difference between fake and actual values. We can optimize the generator depending on what fools and does not fool the discriminator

For generator, We map the latent space to input space $G : \mathcal{Z} \to \mathbb{R}^n, z \mapsto \hat{x}$

For discriminator, we go from the original input (x) or the crafted input ($\hat{x}$) to a learned loss. $D : \mathbb{R}^n \to (0, 1) \quad x \vee \hat{x} \mapsto \ell$

34.2.2 Model illustration

We start from a random variable in the hidden space. Then we get our $\hat{x}$ from the generator. Note that in the GAN setting, we do not know our data manifold, so there will be no way trying to proximate $\hat{x}$ to data manifold. We apply our discriminator on both the crafted data and real data.

We train our discriminator so that it can tell apart the two. On the other hand, we get the gradient, so that we can make the generated data as close to the real data manifold as possible.

When training GAN, we don't use a so called loss function because we not not trying to minimizing it. Instead, we use the following value function.

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

$$\min_G \max_D V(D,G)$$

The above function reaches an equilibrium (Nash equilibrium) between the performance of the discriminator and the performance of the generator. When the two modules reach their best performance together, the discriminator should be able to tell apart fake data and real data, and the generator will be generating perfect data.

Part III

Coding

Chapter 35

Multimodule Systems

35.1 Multi-layer Neural Network

In section 4.3, we discussed a forward multimodule system. The following code includes three different ways of building a two-layer neural network.

```
import torch
from torch import nn

image = torch.randn(3, 10, 20)
in_size = image.nelement()
h_size = 60
out_size = 6

#### Functional paradigm
m1 = nn.Linear(in_size, h_size)
m2 = nn.Linear(h_size, out_size)
# forward prop
hid = torch.relu(m1(image.view(-1)))
out = m2(hid)

#### Using containers
model = nn.Sequential(m1, nn.relu(), m2)
# forward prop
out = model(image.view(-1))

#### Using object oriented programming
class Net(nn.Module):
    def __init__(self, in_s, h_s, out_s):
        super().__init__()
        self.m1 = nn.Linear(in_s, h_s)
        self.m2 = nn.Linear(h_s, out_s)
```

```
        def forward(self, x):
            x = torch.relu(self.m1(x.view(-1)))
            x = self.m2(x)
            return x

    model = Net(in_size, h_size, out_size)
    out = model(image)
```

First, we import torch and the `nn` module from torch in Python. The `nn` module has several predefined modules. The input is a random matrix of size $3 \times 10 \times 20$. We can think of this matrix as an image with 3 RGB components and is 10 rows by 20 columns. The total size is obtained by `image.nelement()`. Next, we build a neural network with two linear layers, which are multiplications by matrix.

Using functional paradigm, we create the first module `m1` using `nn.Linear()` and give the sizes of the input `in_size` and output `h_size`. Similarly, the second module `m2`, which is also a linear module, is created with `nn.Linear()`. It takes the vector of the same size `h_size` and produces the vector of the output size `out_size`, which is 6 (a 6-way classification). Then, the forward propagation calls functions `m1` and `m2`. The variable `hid` first applies the module `m1` to `image.view(−1)`, that is, it takes the `image` and uses `view.(−1)` to turn a 3-dimensional tensor into a single vector. Then, `hid` applies `ReLU` (Rectifier Linear Unit) to the single vector. Each component of the vector is passed through a halfway rectifier. Recall that the output of the `ReLU` function is the identity function if the argument is positive and 0 if the argument is negative. Lastly, we obtain the result `out` by taking the result `hid` and applying the second module `m2` to it.

Another way of building a neural network is to use containers that define certain predefined structures. Instead of writing functions individually for each module, variable `model` uses a container, `nn.Sequential()`, to build a graph and pass the signal in the order of the input modules `m1`, `nn.ReLU()`, and `m2`. Note that the list of modules is called a sequence. Then, the forward propagation obtains the result `out` by taking `model` and applying `image.view(−1)` to it.

Lastly, we can define a class using object oriented programming for this particular two-layer neural network. First, we initiate parameters `in_s`, `h_s`, and `out_s` for input and output sizes. Then, we create two linear module `m1` and `m2` using `nn.Linear()`. Next, the forward function first takes the input `x` and turns it into a single vector. Then, the function applies `m1` module to the vector and applies `ReLU` to the result. The module `m2` is then applied to the result `x`. The variable `model` then calls the class `Net` and the variable `out` applies the `image` to the class to obtain the final result. Note that this method is already being implemented by PyTorch.

When we run a feedforward neural network, PyTorch automatically calculates the gradient of the weight using backpropagation.

Chapter 36

Introduction to *PyTorch* and `Tensors`

36.1 What is *PyTorch*?

PyTorch is a Python based scientific computing package targeting on two sets of audiences:

- Tensorial library that uses the power of GPUs
- A deep learning research platform that provides maximum flexibility and speed

```
In [1]: import torch
```

36.2 Getting help in Jupyter

Jupyter Notebook is a common Integrated Development Environment (IDE) for deep learning. There are a few commands specific to Jupyter Notebook that are helpful for coding.

36.2.1 Using tab

Tab will list all available functions, while shift + tab will open the documentation.

36.2.2 Using ?

```
In [2]: # Open the documentation, same as <shift> + <tab> on 'torch.nn.Module()'
        torch.nn.Module?
```

```
In [3]: # See the source code of all functions being executed in the Module
        torch.nn.Module??
```

36.2.3 Dropping to Bash

```
In [4]: # List all the files in the current directory
        !ls -lh
```

```
In [5]: # Getting some general help
        %magic
```

36.3 Tensors

A tensor is a n-dimensional array, *PyTorch* provides functions for operating on Tensors like numpy does for arrays.

```
In [6]: # Generate a tensor of size 2x3x4
        t = torch.Tensor(2, 3, 4)
        type(t)

Out[6]: torch.Tensor

In [7]: # Get the size of the tensor
        t.size()

Out[7]: torch.Size([2, 3, 4])

In [8]: # Get the dimension of the tensor, for example, 1 for vectors, 2 for matrices
        t.dim()

Out[8]: 3

In [9]: # Total number of elements in the tensor
        t.numel()

Out[9]: 24
```

Note: Mind the underscore! Any operation that mutates a tensor in-place is post-fixed with an underscore. The in-place replacement will change the object. It is encouraged to perform operations in-place to optimize usage of memory.

```
In [10]:
        t.random_(10)

Out[10]: tensor([[[4., 6., 2., 4.],
                  [8., 2., 6., 9.],
                  [1., 4., 9., 9.]],

                 [[8., 5., 5., 5.],
                  [4., 1., 1., 7.],
                  [4., 4., 6., 9.]]])

In [11]: r = torch.Tensor(t)
         # This resizes the tensor permanently
         r.resize_(3, 8)
         r

Out[11]: tensor([[4., 6., 2., 4., 8., 2., 6., 9.],
                 [1., 4., 9., 9., 8., 5., 5., 5.],
                 [4., 1., 1., 7., 4., 4., 6., 9.]])
```

```
In [12]: # Replace all element in r with 0's
         r.zero_()

Out[12]: tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0., 0., 0., 0.]])

In [13]: t

Out[13]: tensor([[[0., 0., 0., 0.],
                  [0., 0., 0., 0.],
                  [0., 0., 0., 0.]],

                 [[0., 0., 0., 0.],
                  [0., 0., 0., 0.],
                  [0., 0., 0., 0.]]])

In [14]: # Make a copy of r rather than replace r.
         s = r.clone()
```

Q: Why don't we always do this?
A: It's time-consuming due to memory allocation, especially when we are training a neural network.

```
In [15]: # In-place fill of 1's
         s.fill_(1)
         s

Out[15]: tensor([[1., 1., 1., 1., 1., 1., 1., 1.],
                 [1., 1., 1., 1., 1., 1., 1., 1.],
                 [1., 1., 1., 1., 1., 1., 1., 1.]])

In [16]: # Because we cloned r, even though we did an in-place operation, this doesn't affect r
         r

Out[16]: tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0., 0., 0., 0.]])
```

36.3.1 Vectors: 1D Tensor

```
In [17]: v = torch.Tensor([1, 2, 3, 4])
         w = torch.Tensor([1, 0, 2, 0])
         # Element-wise multiplication
         v * w

Out[17]: tensor([1., 0., 6., 0.])

In [18]: # Scalar product: 1*1 + 2*0 + 3*2 + 4*0
         v @ w
```

```
Out[18]: tensor(7.)

In [19]: x = torch.Tensor(5).random_(10)
         x

Out[19]: tensor([6., 0., 5., 9., 7.])

In [20]: # Extract sub-Tensor [from:to)
         x[1:2 + 1]

Out[20]: tensor([0., 5.])
```

Note: torch.arange gives only integers. Use torch.arange(1, 4 + 1, dtype = torch.float) to generate float numbers.

```
In [21]: # Create a tensor with integers ranging from 1 to 5, excluding 5
         v = torch.arange(1, 4 + 1)
         v

Out[21]: tensor([1, 2, 3, 4])
```

36.3.2 Matrices: 2D Tensor

```
In [22]: # Create a 2x4 tensor
         m = torch.Tensor([[2, 5, 3, 7],
                           [4, 2, 1, 9]])

In [23]: # Indexing column 0, row 2, it returns a 0-dimensional tensor, which is a scalar. Or m[0,2]
          m[0][2]

Out[23]: tensor(3.)

In [24]: # Extract the single value in the scalar
          m[0,2].item()
```

Q: Why won't we extract value always?
A: A tensor will remember who is the parent, so we need to use tensor rather than just the scalar number when we perform back propagation during training a neural network.

```
Out[24]: 3.0

In [25]: # Indexing column 1, all rows (returns size 2)
          m[:, 1]

Out[25]: tensor([5., 2.])

In [26]: # Add one more bracket inside will add an extra dimension, now it's a 2 * 1 matrix
          m[:, [1]]

Out[26]: tensor([[5.],
                 [2.]])
```

```
In [27]: # Same as m.transpose(0, 1)
         m.t()
```

Note: We can specify dimensions dimensions we want to swap using m.transpose(c1,c2) when we have many dimensions.

```
Out[27]: tensor([[2., 4.],
                 [5., 2.],
                 [3., 1.],
                 [7., 9.]])

In [28]: # Create tensor from 3 to 8, with each having a space of 1
         torch.arange(3., 8 + 1)

Out[28]: tensor([3., 4., 5., 6., 7., 8.])

In [29]: # returns a 1D tensor of steps equally spaced points between start=3, end=8 and steps=20
         torch.linspace(3, 8, 20).view(1, -1)

Out[29]: tensor([[3.0000, 3.2632, 3.5263, 3.7895, 4.0526, 4.3158, 4.5789, 4.8421, 5.1053,
                  5.3684, 5.6316, 5.8947, 6.1579, 6.4211, 6.6842, 6.9474, 7.2105, 7.4737,
                  7.7368, 8.0000]])

In [30]: # Create a tensor filled with 0's torch.ones() will create a tensor filled with 1's.
         torch.zeros(3, 5)

Out[30]: tensor([[0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0.]])

In [31]: # Create a tensor with the diagonal filled with 1
         torch.eye(3)

Out[31]: tensor([[1., 0., 0.],
                 [0., 1., 0.],
                 [0., 0., 1.]])
```

36.3.3 Images: 3D Tensor

Q: Why would we use 4D Tensors while training convolutional neural networks?
A: Images are 3D Tensors with dimensions channels, height, and width. We will use a stack of multiple images to form a 4-dimensional tensor.

36.4 References

- matrices-as-transformations.shtml
- 01-tensor_tutorial.ipynb
- 02-space_stretching.ipynb

Chapter 37

Spiral Classification

Remember, n connected linear transformations is still a linear transformation.

Let's review some of the transformations we can perform on our data. Linear transformations (matrices), can scale, rotate, and shear. In order to 'squash', we need non-linearities.

Now, let's see how we can use PyTorch to perform classification.

37.1 Creating the Data

First, we need to import the necessary libraries:

```
import random
import torch
from torch import nn, optim
import math
from IPython import display

from plot_lib import plot_data, plot_model, set_default
```

Next, we need to set a constant value seed, so all random initializations are the same for all of us. We will also set up the dimensions of our problem. The entire network will map from 2D to 3D as the input layer is D, and the output layer will correspond to the number of classes, C. C also corresponds to the 3 possible colors we can classify each point as.

```
seed = 12345
random.seed(seed)
torch.manual_seed(seed)
N = 1000  # num_samples_per_class
D = 2  # dimensions
C = 3  # num_classes
H = 100  # num_hidden_units
```

As we see above, we have H with a value of 100. This means we are going to map from our 2D input space to 100 dimensions. We do this so we can make the input easier to separate. We will then map it back down to 3D.

Please note that the value of each of the 3 dimensions in the output will be the probability the point is in that class.

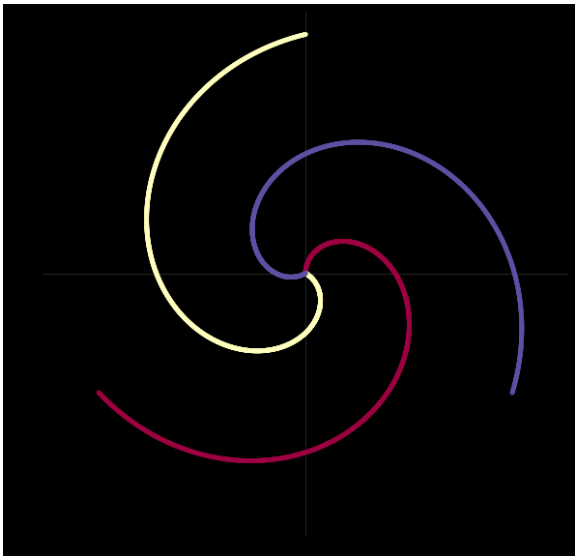
Why do we even bother mapping from a low dimensional input space to higher dimensions? Well, it "disentangles" the input space so we can linearly separate it.

Remember, we want to think of our network as having a top and a bottom. We want to think of the input space on the bottom and the output space at the top. Therefore, we want our data to be linearly separable at the top, or the end of the network.

Before we begin training any nets on our data, we have to generate some! The code below does just that:

```
X = torch.zeros(N * C, D)
y = torch.zeros(N * C, dtype=torch.long)
for c in range(C):
    index = 0
    t = torch.linspace(0, 1, N)
    # When c = 0 and t = 0: start of linspace
    # When c = 0 and t = 1: end of linpace
    # This inner_var is for the formula inside sin() and cos()
    # like sin(inner_var) and cos(inner_Var)
    inner_var = torch.linspace(
        # When t = 0
        (2 * math.pi / C) * (c),
        # When t = 1
        (2 * math.pi / C) * (2 + c),
        N
    ) + torch.randn(N) * 0

    for ix in range(N * c, N * (c + 1)):
        X[ix] = t[index] * torch.FloatTensor((
            math.sin(inner_var[index]), math.cos(inner_var[index])
        ))
        y[ix] = c
        index += 1

print("Shapes:")
print("X:", tuple(X.size()))
print("y:", tuple(y.size()))
```

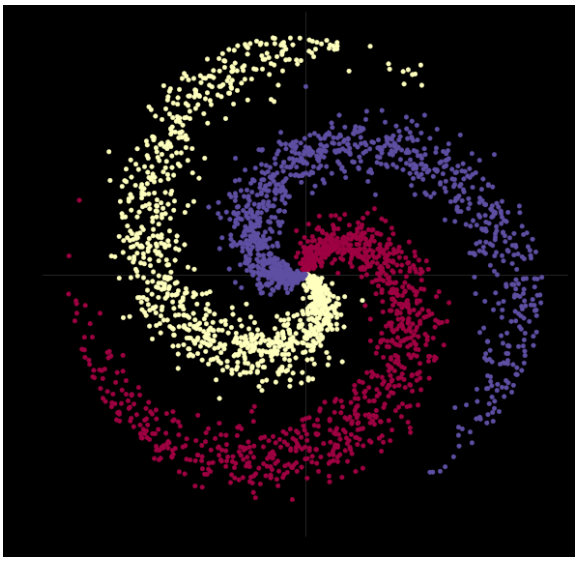
Let's view this data:



Note that this line:

```
torch.randn(N) * 0
```

determines how much noise we could add to the data. If we change the value of 0 to be let's say .33, we get this plot:



37.2 Linear Model

Now, let's create a simple neural network with only linear layers.

```
learning_rate = 1e-3
lambda_l2 = 1e-5

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# nn package to create our linear model
# each Linear module has a weight and bias
model = nn.Sequential(
    nn.Linear(D, H),
    nn.Linear(H, C)
)
model.to(device) #Convert to CUDA

# nn package also has different loss functions.
# we use cross entropy loss for our classification task
criterion = torch.nn.CrossEntropyLoss()

# we use the optim package to apply
# stochastic gradient descent for our parameter updates
# built-in L2
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay=lambda_l2)

# Training
for t in range(1000):

    # Feed forward to get the logits
    y_pred = model(X)

    # Compute the loss and accuracy
    loss = criterion(y_pred, y)
    score, predicted = torch.max(y_pred, 1)
    acc = (y == predicted).sum().float() / len(y)
    print("[EPOCH]: %i, [LOSS]: %.6f, [ACCURACY]: %.3f" % (t, loss.item(), acc))
    display.clear_output(wait=True)

    # zero the gradients before running
    # the backward pass.
    optimizer.zero_grad()

    # Backward pass to compute the gradient
    # of loss w.r.t our learnable params.
    loss.backward()

    # Update params
    optimizer.step()
```

First, we set up some hyper parameters and then move the model to the GPU's memory if

PyToch detects a GPU. This model will move our data from D dimensions to H dimensions, and then back to C.

As you can see, this model uses CrossEntropyLoss to judge the performance of the classifier. It also uses SGD as the optimizer. Remember, we prefer stochastic gradient descent over batch gradient descent because most data has a lot of redundancy and we do not need to use every data point to compute the gradient.

epoch: One full pass through the entire training data. We will sweep over every training example here 1000 times. That is, we will be running 1000 epochs.

Why do we have to zero the gradients before the backward pass? PyTorch automatically accumulates the gradients as it goes through the network. If we don't zero out the gradients, we keep adding new gradients to dirty memory which we don't want. We want to create a fresh new gradient before each step.

Why do we add gradients to begin with? If we keep adding gradients, we end up with the average gradient over all the training data which is what we would end up with anyways with batch gradient descent.

NB: Always remember to zero your gradients! This is easily forgotten.

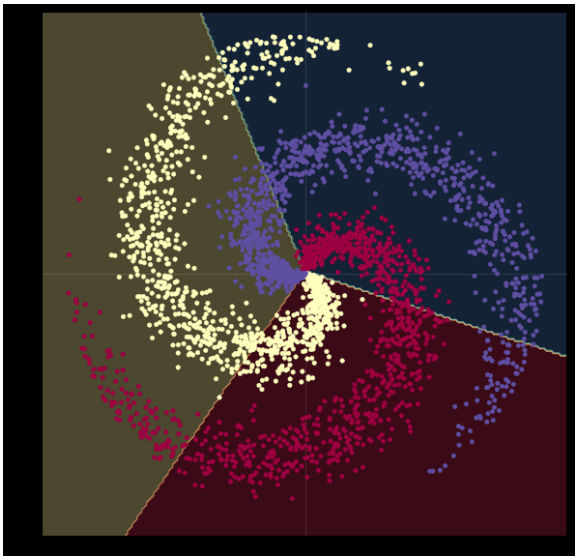
What happens if we invert these two lines ?

```
optimizer.zero_grad()
loss.backward()
```

We'd get nothing as the gradient would always be zero so our model would take no steps.

What is optimizer.step()? This line uses the gradient to find the best direction to change the parameter weights. It finds the best direction and takes a "step" in that direction to values that give better loss.

After running this model, we get the following decision boundaries, visualized here:



What's wrong with this model? It's only giving about 50% accuracy!

Remember, with 3 classes, random guessing would give us 33.33% accuracy so we are still doing somewhat better than chance. This is because the model can do some work by simply rotating the decision boundaries.

NB: Always print the initial loss before any optimizer steps. This is because we need to see if our model is actually improving at all from a random initialization of weights.

Here is how to compute the initial loss for the model we have above. With three classes, we can compute what our initial loss would be by using

```
-math.log(1/3) # 1/3 because of random guess for 3 classes
# or you can equivalently use:
math.log(3) # outputs 1.09861228867
```

This value is what we would see with using a random guess for one of each classes. As we saw from running the notebook, with a Linear Model, we can achieve a final loss of 0.861541 which is lower than initial loss of 1.0986. Computing what the initial loss would have been with random guess and putting that into the loss function is very important because it allows us to see that after some training, we did in fact do 'better.'

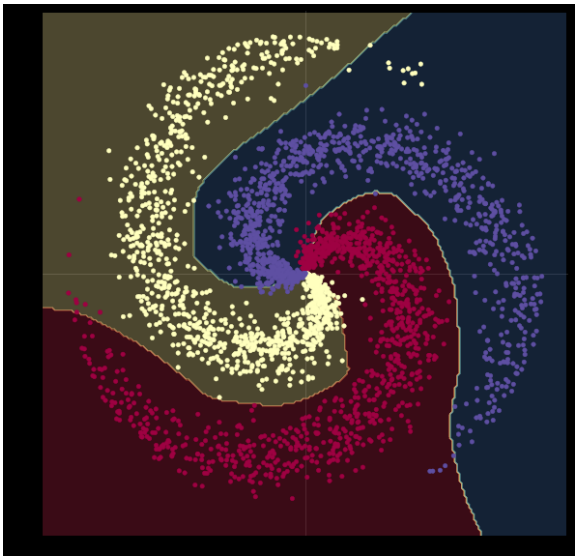
37.3 Non-Linear Model

Now, let's add a ReLu in between the two linear layers and see what we get.

```
learning_rate = 1e-3
lambda_l2 = 1e-5

model = nn.Sequential(
    nn.Linear(D, H),
    nn.ReLU(),
    nn.Linear(H, C)
)
model.to(device)

# nn package also has different loss functions.
# we use cross entropy loss for our classification task
criterion = torch.nn.CrossEntropyLoss()

# we use the optim package to apply
# ADAM for our parameter updates
# built-in L2
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=lambda_l2)

# e = 1.  # plotting purpose

# Training
for t in range(1000):

    # Feed forward to get the logits
```

```
y_pred = model(X)

# Compute the loss and accuracy
loss = criterion(y_pred, y)
score, predicted = torch.max(y_pred, 1)
acc = (y == predicted).sum().float() / len(y)
print("[EPOCH]: %i, [LOSS]: %.6f, [ACCURACY]: %.3f" % (t, loss.item(), acc))
display.clear_output(wait=True)

# zero the gradients before running
# the backward pass.
optimizer.zero_grad()

# Backward pass to compute the gradient
# of loss w.r.t our learnable params.
loss.backward()

# Update params
optimizer.step()
```

WOW: after running this, we get a model that classifies with 0.934 accuracy.
How does it look? Very pretty as you can see for yourselves below:



Where are we looking from in these visualizations? From the bottom up! This is because the shape of the data we see here is before any transformations are done on it.

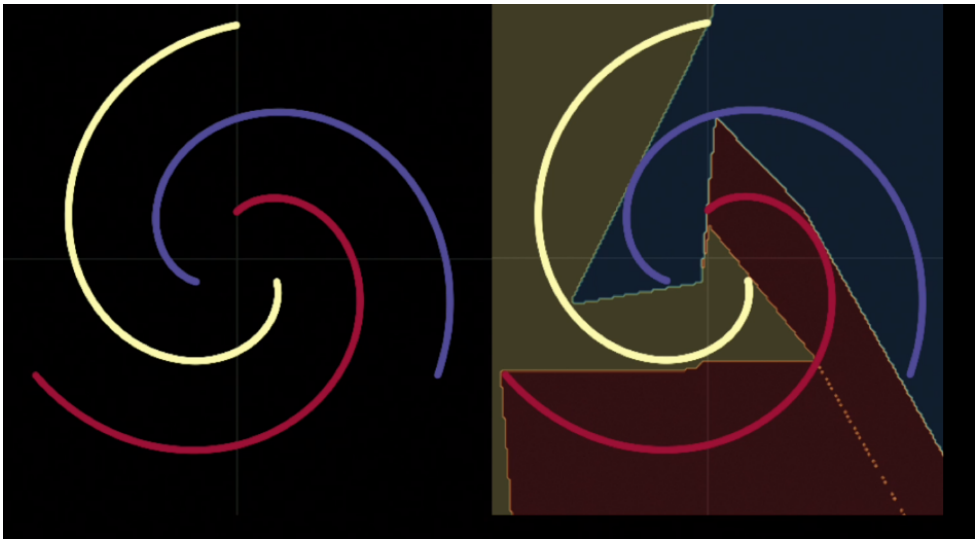


Figure 37.1: No projection to a higher dimension

37.4 Skinny Model

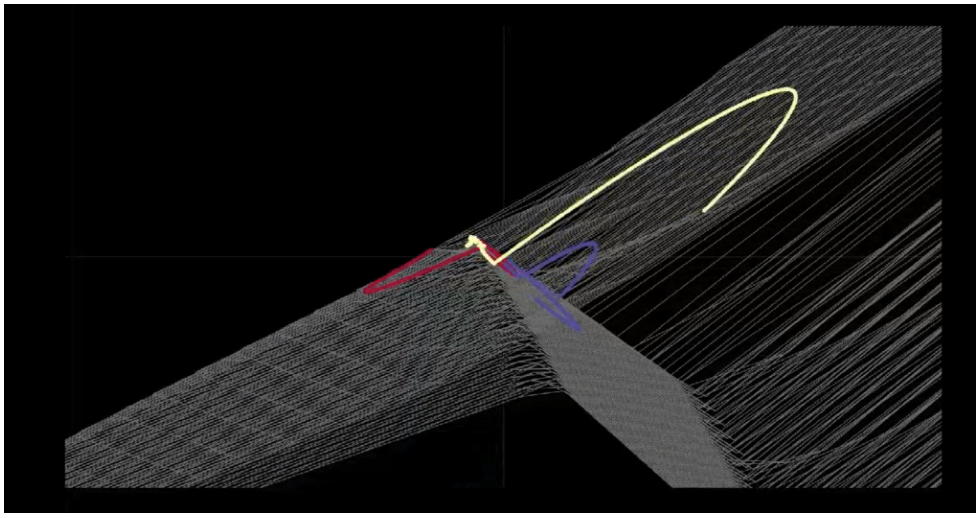
Now, let's remove the noise of same input data. Let's use a neural network that doesn't expand the input to 100 dimensions this time and see what kind of classifier we get.

Here is the code that was used to generate a decision boundary like this:

```
# NN model
def two_layer_network(D_in, D_out, hidden_layers, H):

    modules = list()
    in_d = D_in

    for l in range(hidden_layers):
        modules.append(nn.Linear(in_d, H))
        modules.append(nn.LeakyReLU())
        in_d = H
    modules.append(nn.Linear(H, H))  # added layer
    modules.append(nn.Linear(H, D_out))
    return nn.Sequential(*modules)

learning_rate = 1e-2
lambda_l2 = 1e-9

# D = 2, C = 3
model = two_layer_network(D, C, 4, D)

# ...use the same code as seen above to train this model
```

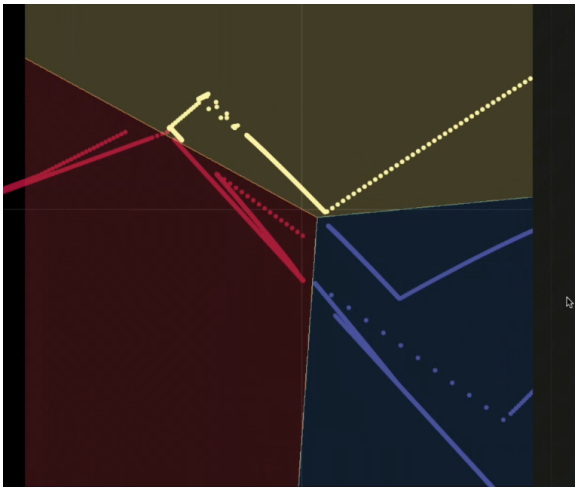
The final decision boundary is very ugly and rigid. This is because we did not expand the dimensions of the data. This limits how much we can stretch out the plane to then divide the data. However, with enough tweaking of the random seed and rerunning the model, even this constrained model can achieve 100% accuracy on this data as seen from figure 37.1.

Let's see the entire dimension space transformed by this network:

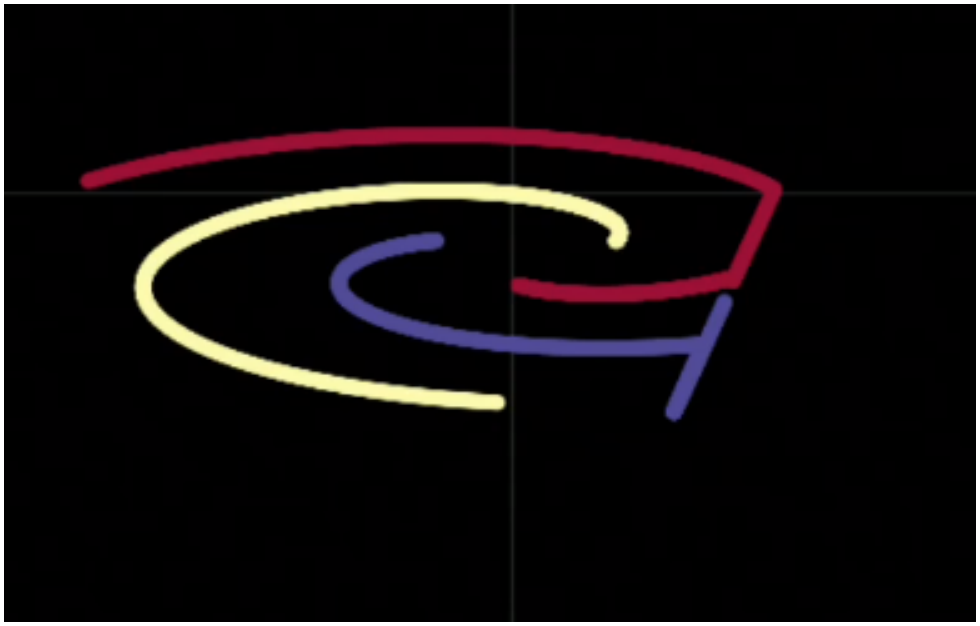


The network actually warps the entire space to make the spirals linearly separable.

Now, we have seen the decision boundary from the bottom of the network. Let's see what the decision boundary looks like from the top of the network:



If you visualize the space after each transformation, you can actually see the ReLu 'cut off' negative points and 'squash' them to 0. Here is an example of some of the points 'squashed' by the ReLu after one step:



In short, the network's job is simply to warp space until we attach a final layer that does a simple classification on the new, easily separated space.

How do you know this sequence will make this space linearly separable?

You visually see that the space can be separated by some line. They aren't even overlapped so it should be easy given some specific stretching.

One thing to notice is that if you do map to a higher dimension and then back down to the output, you will see smoother decision boundaries. The decision boundary we saw with this non-expansive model is more piece-wise.

If you use tanh or sigmoid non-linearities, the transitions from each new transformed dimension space are much smoother than if you clip and 'squash' with ReLu.

As Professor LeCun stated, we get these transformations because we are letting the optimizer decide what the next best transformation to run is that minimizes the loss.

What is bottom up vs top down when discussing networks?

Input is bottom, output is top (because you climb up hierarchy). If you draw the network upside down, it makes an inverted representation. So, when we say we add classifier on top, it means we add it to the last layer in the network.

How does ReLu divide the input into subspaces?

This splitting actually applies to any non-linearity. Any time you have one in the activation function, you're basically performing elementary detection. So, imagine a threshold function. You're basically dividing the input into what section outputs the 1 and what section outputs the -1. However, simple threshold functions destroy too much information so we look for softer threshold functions. ReLu is an example of a softer threshold that preserves more information.

In addition to the ReLu, you can also use a Leaky ReLu: A Practical Guide to ReLU

According to Professor LeCun, small networks are good to visualize what is going on but will fail for most problems. This spiral problem requires very little width in the network so that's why it works.

NB: The bigger the network, the easier it is to train as gradient descent has more time to optimize.

In addition to this note, the networks that expand to higher dimensions are trained much better and result in smoother decision boundaries. These boundaries are still piece-wise linear if you zoom in enough, but its almost impossible to see it from the output view.

Chapter 38

Regression

The following code shows some experiments on how to use neural networks for regression.

38.1 Create the data

```
import random
import torch
from torch import nn, optim
import math
from IPython import display
from plot_lib import plot_data, plot_model, set_default
from matplotlib import pyplot as plt

set_default()

seed = 12345
random.seed(seed)
torch.manual_seed(seed)
N = 1000  # num_samples_per_class
D = 1  # dimensions
C = 1  # num_classes
H = 100  # num_hidden_units
X = torch.unsqueeze(torch.linspace(-1, 1, 100), dim=1)
y = X.pow(3) + 0.3 * torch.rand(X.size())

print("Shapes:")
print("X:", tuple(X.size()))
print("y:", tuple(y.size()))

plt.scatter(X.numpy(), y.numpy())
plt.axis('equal')
```

Since we want to do a Regression, we treat the data as one-dimensional (D = 1) and the output as one-dimensional (C = 1).

38.2 Linear Model

```
learning_rate = 1e-3
lambda_l2 = 1e-5

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# nn package to create our linear model
# each Linear module has a weight and bias
model = nn.Sequential(
    nn.Linear(D, H),
    nn.Linear(H, C)
)
model.to(device) # Convert to CUDA
# nn package also has different loss functions.
# we use MSE loss for our regression task
criterion = torch.nn.MSELoss()
# we use the optim package to apply
# stochastic gradient descent for our parameter updates
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay=lambda_l2) # built-in

for t in range(1000):
    # Feed forward to get the logits
    y_pred = model(X)
    # Compute the loss (MSE)
    loss = criterion(y_pred, y)
    print("[EPOCH]: %i, [LOSS or MSE]: %.6f" % (t, loss.item()))
    display.clear_output(wait=True)
    # zero the gradients before running
    # the backward pass.
    optimizer.zero_grad()
    # Backward pass to compute the gradient
    # of loss w.r.t our learnable params.
    loss.backward()
    # Update params
    optimizer.step()

plt.scatter(X.data.numpy(), y.data.numpy())
plt.plot(X.data.numpy(), y_pred.data.numpy(), 'r-', lw=5)
plt.axis('equal');
```

When using the simple linear model, there are two `nn.Linear` layers without any non-linearity, so this is equivalent to a single linear layer and the output is a straight line. This is called Linear Regression.

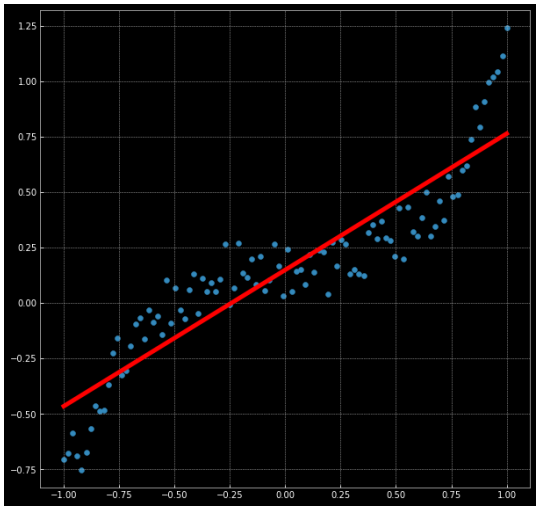


Figure 38.1: Regression using linear NN model (without non-linearity).

38.3 Two-layered network

```
learning_rate = 1e-3
lambda_l2 = 1e-5
# Number of networks
n_networks = 5
models = list()
y_pretrain = list()
# nn package also has different loss functions.
# we use MSE for a regression task
criterion = torch.nn.MSELoss()

for mod in range(n_networks):
    # nn package to create our linear model
    # each Linear module has a weight and bias
    model = nn.Sequential(
        nn.Linear(D, H),
        nn.ReLU(),
        nn.Linear(H, C)
    )
    model.to(device)
    # Append models
    models.append(model)
    # we use the optim package to apply
    # ADAM for our parameter updates
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=lambda_l2) # bu
```

```
    # e = 1.  # plotting purpose
    # Training
    for t in range(1000):
        # Feed forward to get the logits
        y_pred = model(X)
        # Append pre-train output
        if t == 0:
            y_pretrain.append(y_pred)
        # Compute the loss and accuracy
        loss = criterion(y_pred, y)
        print(f"[MODEL]: {mod + 1}, [EPOCH]: {t}, [LOSS]: {loss.item():.6f}")
        display.clear_output(wait=True)
        # zero the gradients before running
        # the backward pass.
        optimizer.zero_grad()
        # Backward pass to compute the gradient
        # of loss w.r.t our learnable params.
        loss.backward()
        # Update params
        optimizer.step()
```

38.4 Predictions: Before Training

```
for y_pretrain_idx in y_pretrain:
    # New X that ranges from -5 to 5 instead of -1 to 1
    X_new = torch.unsqueeze(torch.linspace(-2, 2, 100), dim=1)

    plt.plot(X_new.data.numpy(), y_pretrain_idx.data.numpy(), 'r-', lw=1)

plt.scatter(X.data.numpy(), y.data.numpy())
plt.axis('equal')
```

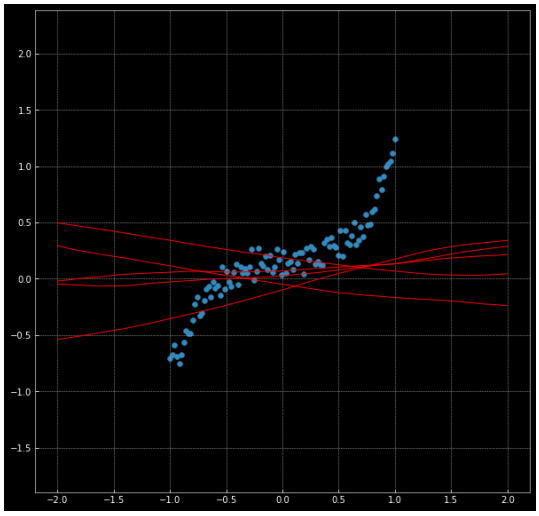


Figure 38.2: Random weights at first.

38.5 Predictions: After Training

```
for model in models:
    # New X that ranges from -5 to 5 instead of -1 to 1
    X_new = torch.unsqueeze(torch.linspace(-1, 1, 100), dim=1)

    # Getting predictions from input
    with torch.no_grad():
        y_pred = model(X_new)

    plt.plot(X_new.data.numpy(), y_pred.data.numpy(), 'r-', lw=1)
plt.scatter(X.data.numpy(), y.data.numpy())
```

When we add a ReLU between the two `nn.Linear` layers, the model now is equivalent to having segments of different straight lines, "selected" by the ReLU activation. Now we can model more complex functions by using these segments; the more hidden units (i.e. ReLU units), the more complex (i.e. more edges) the final function will be. This is called Piecewise-linear Regression.

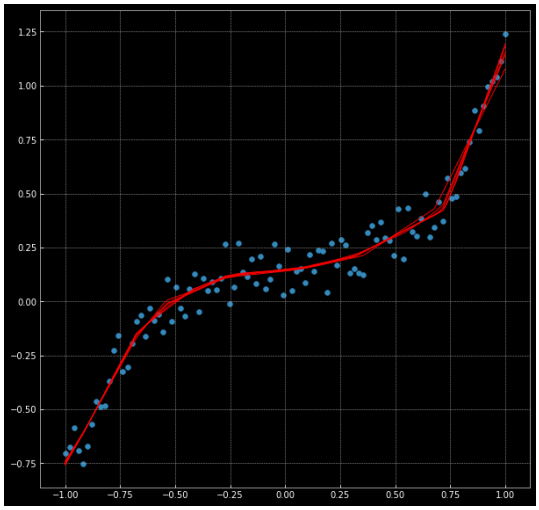


Figure 38.3: Regression using piecewise-linear model (with ReLU non-linearity).

Out of the interval of the train data the variance explodes. We have no information as to what should happen there, so the model does not introduce any new edge and the slopes of the last segments (to the left and right of the data) is extended.

Chapter 39

Convolutional Networks

In this chapter, two models are presented:

- A fully-connected network
- A convolutional network with the same number of parameters as the fully-connected network.

Through two experiments, we show that:

- The ConvNet makes better use of its parameters by exploiting the properties of natural data, thus achieving a higher accuracy than the fully-connected network.
- After random shuffling the pixels of the images in the same way for all inputs, the fully-connected network achieves a similar performance, while the CNN performed worse than the FC network due to the input losing the natural data properties that made convolutions work well.

39.1 Libraries

To run the code in this chapter, we need import all necessary libraries at first. Apart from *PyTorch*, we also need the *torchvision* library. It consists of some popular datasets for computer vision, and also includes some common image transformation functions.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
```

39.2 Dataset

In the chapter, we will use the MNIST handwritten digit database to test our models. The dataset could be imported easily from the *torchvision* library.

```
input_size  = 28 * 28   # images are 28x28 pixels
output_size = 10       # there are 10 classes

# The output of torchvision datasets are PILImage images of range [0, 1].
# We transform them to Tensors of normalized range [-1, 1].
transform = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.1307,), (0.3081,))])
train_set = torchvision.datasets.MNIST(root='./data', train=True, download=True,
    transform=transform)
test_set = torchvision.datasets.MNIST(root='./data', train=False, download=True,
    transform=transform)
```

After getting the MNIST dataset from *torchvision*, we need to create data loaders for training set and test set.

```
train_loader = torch.utils.data.DataLoader(train_set, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=64, shuffle=False)
```

39.3 CNN vs Fully-Connected Networks

We create two image classification models, one only use the linear layers and the other use convolutional layers.

```
class FC2Layer(nn.Module):
    def __init__(self, input_size, n_hidden, output_size):
        super(FC2Layer, self).__init__()
        self.input_size = input_size
        self.network = nn.Sequential(
            nn.Linear(input_size, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, output_size),
            nn.LogSoftmax(dim=1)
        )

    def forward(self, x):
        x = x.view(-1, self.input_size)
        return self.network(x)

class CNN(nn.Module):
    def __init__(self, input_size, n_feature, output_size):
        super(CNN, self).__init__()
        self.n_feature = n_feature
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=n_feature, kernel_size=5)
        self.conv2 = nn.Conv2d(n_feature, n_feature, kernel_size=5)
```

```
        self.fc1 = nn.Linear(n_feature*4*4, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x, verbose=False):
        x = self.conv1(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        x = x.view(-1, self.n_feature*4*4)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.log_softmax(x, dim=1)
        return x
```

Then, we can define the corresponding train and test functions for both models. The `perm` parameter will be used to permute pixels in the second experiment.

```
def train(epoch, model, perm=torch.arange(0, 784).long()):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        # permute pixels
        data = data.view(-1, 28*28)
        data = data[:, perm]
        data = data.view(-1, 1, 28, 28)

        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 100 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f})] tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))

def test(model, perm=torch.arange(0, 784).long()):
    model.eval()
    test_loss = 0
    correct = 0
    for data, target in test_loader:
        # permute pixels
        data = data.view(-1, 28*28)
        data = data[:, perm]
        data = data.view(-1, 1, 28, 28)
```

```
        output = model(data)
        test_loss += F.nll_loss(output, target, reduction='sum').item() # sum up batch loss
        pred = output.data.max(1, keepdim=True)[1] # get the index of the max log-probability
        correct += pred.eq(target.data.view_as(pred)).cpu().sum().item()

    test_loss /= len(test_loader.dataset)
    accuracy = 100. * correct / len(test_loader.dataset)
    print('Test set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)'.format(
        test_loss, correct, len(test_loader.dataset),
        accuracy))
```

39.3.1 Experiment 1: Regular Images

In the first experiment, we will train both models with regular images. The CNN model:

```
# Training settings
n_features = 6 # number of feature maps

model_cnn = CNN(input_size, n_features, output_size)
optimizer = optim.SGD(model_cnn.parameters(), lr=0.01, momentum=0.5)

for epoch in range(0, 1):
    train(epoch, model_cnn)
    test(model_cnn)
```

The result:

```
Train Epoch: 0 [0/60000 (0)] tLoss: 2.320940
Train Epoch: 0 [6400/60000 (11)] tLoss: 2.251726
Train Epoch: 0 [12800/60000 (21)] tLoss: 0.926804
Train Epoch: 0 [19200/60000 (32)] tLoss: 0.602818
Train Epoch: 0 [25600/60000 (43)] tLoss: 0.638813
Train Epoch: 0 [32000/60000 (53)] tLoss: 0.562240
Train Epoch: 0 [38400/60000 (64)] tLoss: 0.189035
Train Epoch: 0 [44800/60000 (75)] tLoss: 0.381978
Train Epoch: 0 [51200/60000 (85)] tLoss: 0.202108
Train Epoch: 0 [57600/60000 (96)] tLoss: 0.393331
Test set: Average loss: 0.2206, Accuracy: 9347/10000 (93%)
```

The fully-connected network:

```
n_hidden = 8 # number of hidden units

model_fnn = FC2Layer(input_size, n_hidden, output_size)
optimizer = optim.SGD(model_fnn.parameters(), lr=0.01, momentum=0.5)
```

```
for epoch in range(0, 1):
    train(epoch, model_fnn)
    test(model_fnn)
```

The result:

```
Train Epoch: 0 [0/60000 (0)] tLoss: 2.352721
Train Epoch: 0 [6400/60000 (11)] tLoss: 1.887923
Train Epoch: 0 [12800/60000 (21)] tLoss: 1.216436
Train Epoch: 0 [19200/60000 (32)] tLoss: 0.964433
Train Epoch: 0 [25600/60000 (43)] tLoss: 0.776250
Train Epoch: 0 [32000/60000 (53)] tLoss: 0.598156
Train Epoch: 0 [38400/60000 (64)] tLoss: 0.527515
Train Epoch: 0 [44800/60000 (75)] tLoss: 0.407989
Train Epoch: 0 [51200/60000 (85)] tLoss: 0.417804
Train Epoch: 0 [57600/60000 (96)] tLoss: 0.485479
Test set: Average loss: 0.4804, Accuracy: 8625/10000 (86%)
```

From Experiment 1, we could see that CNN is better than fully-connected network on regular image classification.

39.3.2 Experiment 2: Permuted Pixels

In the second experiment, we permute the pixels in a regular image, thereby losing locality in the image data. In this setup, the CNN should perform worse than itself with regular images, and the performance of fully-connected network should not have too much of difference.

```
perm = torch.randperm(784)
for i in range(10):
    image, _ = train_loader.dataset.__getitem__(i)
    # permute pixels
    image_perm = image.view(-1, 28*28).clone()
    image_perm = image_perm[:, perm]
    image_perm = image_perm.view(-1, 1, 28, 28)

# Training settings
n_features = 6 # number of feature maps

model_cnn = CNN(input_size, n_features, output_size)
optimizer = optim.SGD(model_cnn.parameters(), lr=0.01, momentum=0.5)

for epoch in range(0, 1):
    train(epoch, model_cnn, perm)
    test(model_cnn, perm)

n_hidden = 8    # number of hidden units
```

```
model_fnn = FC2Layer(input_size, n_hidden, output_size)
optimizer = optim.SGD(model_fnn.parameters(), lr=0.01, momentum=0.5)

for epoch in range(0, 1):
    train(epoch, model_fnn, perm)
    test(model_fnn, perm)
```

The training result for CNN:

```
Train Epoch: 0 [0/60000 (0)] tLoss: 2.311677
Train Epoch: 0 [6400/60000 (11)] tLoss: 2.282326
Train Epoch: 0 [12800/60000 (21)] tLoss: 2.209269
Train Epoch: 0 [19200/60000 (32)] tLoss: 2.039113
Train Epoch: 0 [25600/60000 (43)] tLoss: 1.641617
Train Epoch: 0 [32000/60000 (53)] tLoss: 1.230703
Train Epoch: 0 [38400/60000 (64)] tLoss: 0.967219
Train Epoch: 0 [44800/60000 (75)] tLoss: 0.741154
Train Epoch: 0 [51200/60000 (85)] tLoss: 0.711306
Train Epoch: 0 [57600/60000 (96)] tLoss: 0.436662
Test set: Average loss: 0.5632, Accuracy: 8229/10000 (82%)
```

The training result for fully-connected network:

```
Train Epoch: 0 [0/60000 (0)] tLoss: 2.280286
Train Epoch: 0 [6400/60000 (11)] tLoss: 1.973936
Train Epoch: 0 [12800/60000 (21)] tLoss: 1.469295
Train Epoch: 0 [19200/60000 (32)] tLoss: 1.052585
Train Epoch: 0 [25600/60000 (43)] tLoss: 0.991563
Train Epoch: 0 [32000/60000 (53)] tLoss: 0.638042
Train Epoch: 0 [38400/60000 (64)] tLoss: 0.472083
Train Epoch: 0 [44800/60000 (75)] tLoss: 0.382082
Train Epoch: 0 [51200/60000 (85)] tLoss: 0.436203
Train Epoch: 0 [57600/60000 (96)] tLoss: 0.561125
Test set: Average loss: 0.4971, Accuracy: 8559/10000 (86%)
```

The results show that convolutional network makes the assumption that pixels lie on a grid are stationary or local. It loses performance when this assumption is wrong. The fully-connected network does not make this assumption. It does less well, or approximately the same, when the assumption is true, since it doesn't take advantage of this prior knowledge.

Chapter 40

Sequence Modeling

In this chapter two applications of sequential networks are present:

1. Sequence Classification - Given character sequences, classify them into pre-specified categories.
2. Signal Echoing - Given an input signal, generate the same signal but with a lag of n steps.

Both of these tasks test the capability of the network to keep track of arbitrary long-term dependencies in the input sequences.

40.1 Sequence Classification

To illustrate the necessity of long-term memory, consider a sequence classification problem. Each input sequence starts with the letter B and ends with E (the trigger symbol). The symbols in-between are randomly chosen from the set {a, b, c, d} except for two elements at positions t_1 and t_2 that are either X or Y. The classification targets are 4 sequence classes Q, R, S and U which depend on the temporal order of X and Y. The rules are

$$\text{X, X} \rightarrow \text{Q}$$
$$\text{X, Y} \rightarrow \text{R}$$
$$\text{Y, X} \rightarrow \text{S}$$
$$\text{Y, Y} \rightarrow \text{U}$$

There are 5 difficulty levels for this problem -

1. EASY : The length of the sequence is randomly chosen from [7, 8]; t_1 is randomly chosen from [1, 2]; t_2 is randomly chosen from [4, 5].
2. NORMAL : The length of the sequence is randomly chosen from [30, 40]; t_1 is randomly chosen from [2, 5]; t_2 is randomly chosen from [20, 27].
3. MODERATE : The length of the sequence is randomly chosen from [60, 80]; t_1 is randomly chosen from [10, 20]; t_2 is randomly chosen from [45, 54].

4. HARD : The length of the sequence is randomly chosen from [100, 110]; t_1 is randomly chosen from [10, 20]; t_2 is randomly chosen from [50, 60].
5. NIGHTMARE : The length of the sequence is randomly chosen from [300, 500]; t_1 is randomly chosen from [10, 80]; t_2 is randomly chosen from [250, 290].

As one can see, long-term memory of X or Y symbols is required to correctly classify the sequence. The HARD difficulty is identical to the original setting of Hochreiter and Schmidhuber [1997]. In the subsequent sections, we will explore the data and train two models. Full details of implementation can be found at Sequence Classification.

40.1.1 Dataset Exploration

Consider sequences of EASY difficulty generated using `get_predefined_generator` method of the `TemporalOrderExp6aSequence` class.

```
from sequential_tasks import TemporalOrderExp6aSequence

# Create a data generator.
example_generator = TemporalOrderExp6aSequence.get_predefined_generator(
    # The first argument is the difficulty level of the classification task.
    difficulty_level=TemporalOrderExp6aSequence.DifficultyLevel.EASY,
    # The second argument is the number of sequences generated in each batch of
    ↪  data.
    batch_size=32,
)
```

Let's print the details of a sample.

```
The return type is a <class 'tuple'> with length 2.
The first item in the tuple is the batch of sequences with shape (32, 9, 8).
The first element in the batch of sequences is:
 [[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0]
 [0 0 0 1 0 0 0 0]
 [1 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [1 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 1]]
The second item in the tuple is the corresponding batch of class labels with shape (32, 4).
The first element in the batch of class labels is:
 [1. 0. 0. 0.]
```

As observed in the output, the sequences are 1-hot encoded in memory. Since there are 8 possible symbols {X, Y, a, b, c, d, B, E}, the last dimension has size 8. The first dimension is the batch size whereas the second dimension corresponds to the length of the sequence. Since length of the sequence is variable, to maintain consistency, the data is zero-padded at the beginning.

Zero-padding is performed at the beginning rather than the end so that back-propagation can be terminated at the start trigger (B).

```
# Decoding the first sequence.
sequence_decoded = example_generator.decode_x(example_batch[0][0, :, :])

# Decoding the class label of the first sequence.
class_label_decoded = example_generator.decode_y(example_batch[1][0])
```

The decoded sequence is given below.

```
The sequence is: BbXcXcbE
The class label is: Q
```

40.1.2 Model Definition

Let's define two models to capture memory - RNN and LSTM.

```
import torch
import torch.nn as nn

# Set the random seed for reproducible results.
torch.manual_seed(1)

class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        # This just calls the base class constructor.
        super().__init__()
        # Neural network layers assigned as attributes of a Module subclass
        # have their parameters registered for training automatically.
        self.rnn = torch.nn.RNN(input_size, hidden_size, nonlinearity='relu',
        ↪  batch_first=True)
        self.linear = torch.nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # The RNN also returns its hidden state but we don't use it.
        # While the RNN can also take a hidden state as input, the RNN
        # gets passed a hidden state initialized with zeros by default.
        x, _ = self.rnn(x)
        x = self.linear(x)
        return x

class SimpleLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.lstm = torch.nn.LSTM(input_size, hidden_size, batch_first=True)
        self.linear = torch.nn.Linear(hidden_size, output_size)
```

```
    def forward(self, x):
        x, _ = self.lstm(x)
        x = self.linear(x)
        return x
```

The prediction is the argmax of the linear layer output. During training, the loss used is Cross Entropy.

40.1.3 Experiment 1 : EASY Difficulty

Let's consider EASY difficulty first. Both the models are trained for 10 epochs. figure 40.1 shows the RNN model performance.

```
# Setup the training and test data generators.
difficulty     = TemporalOrderExp6aSequence.DifficultyLevel.EASY
batch_size     = 32
train_data_gen = TemporalOrderExp6aSequence.get_predefined_generator(difficulty,
↪  batch_size)
test_data_gen  = TemporalOrderExp6aSequence.get_predefined_generator(difficulty,
↪  batch_size)

# Setup the RNN and training settings.
input_size  = train_data_gen.n_symbols
hidden_size = 4
output_size = train_data_gen.n_classes
model       = SimpleRNN(input_size, hidden_size, output_size)
criterion   = torch.nn.CrossEntropyLoss()
optimizer   = torch.optim.RMSprop(model.parameters(), lr=0.001)
max_epochs  = 10

# Train the model.
model = train_and_test(model, train_data_gen, test_data_gen, criterion,
↪  optimizer, max_epochs)

for parameter_group in list(model.parameters()):
    print(parameter_group.size())
```

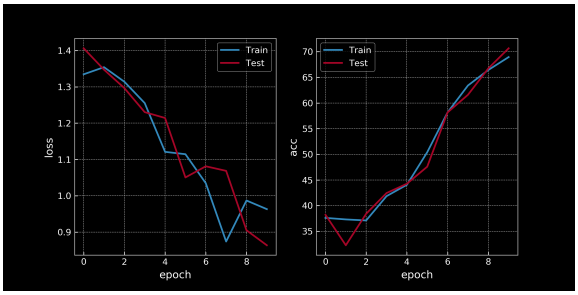


Figure 40.1: EASY Difficulty - RNN Model for 10 epochs

Using 10 epochs, we obtain $\approx 70\%$ in testing accuracy. We observe better testing accuracy than training accuracy during certain periods. The reason for this is that the training accuracy is computed with the model trained using the last batch whereas the testing accuracy is computed with the model trained using the current batch. One should instead compare the current training accuracy with the test accuracy one step back. figure 40.2 shows the performance of LSTM model.

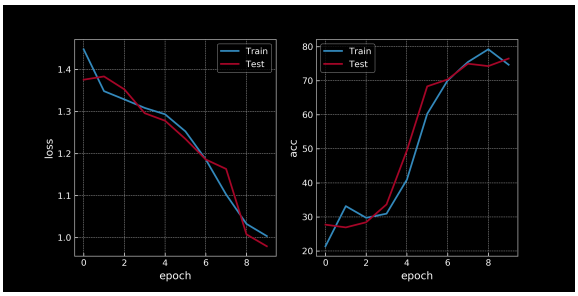


Figure 40.2: EASY Difficulty - LSTM Model for 10 epochs

Since there is scope for further improvement, the models are trained for 100 epochs. figure 40.3 and figure 40.4 show the performances of RNN and LSTM models respectively.

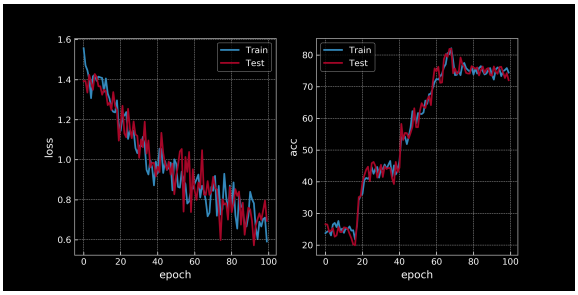


Figure 40.3: EASY Difficulty - RNN Model for 100 epochs

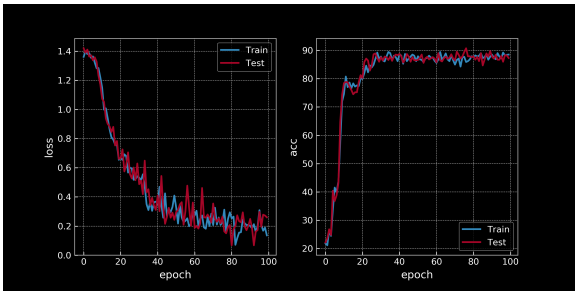


Figure 40.4: EASY Difficulty - LSTM Model for 100 epochs

Both models perform reasonably well and are able to memorize the correct class in EASY difficulty.

40.1.4 Experiment 2 : MODERATE Difficulty

Now let's consider the MODERATE difficulty. The models are trained for 100 epochs and figure 40.5 and figure 40.6 show the performance of the RNN and LSTM models respectively.

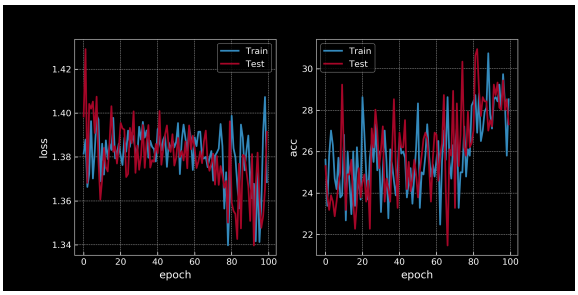


Figure 40.5: MODERATE Difficulty - RNN Model for 100 epochs

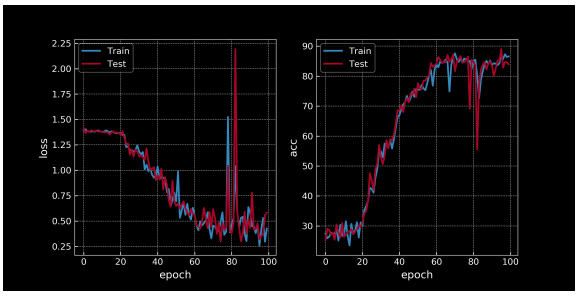


Figure 40.6: MODERATE Difficulty - LSTM Model for 100 epochs

The RNN model performs poorly with a testing accuracy of $\approx 26\%$. This is equivalent to randomly predicting one of the classes. The corresponding loss is ≈ 1.39 which is around $-\ln(0.25)$

as expected.

In comparsion, the LSTM model still achieves accuracy $\approx 84\%$ and thus exhibits long-term memory.

Since LSTM has 4 times the parameters than RNN, this is not a fair comparison. Despite this fact, it sheds light on the ability of LSTM to maintain long-term memory.

40.2 Signal Echoing

In signal echoing, given an input sequence, the task is to regenerate the same sequence with a lag of n steps. This is an example of synchronized many-to-many task as both input and output are sequences. The input is a random binary sequence. Full details about the implementation of this problem can be found at Echo Data.

40.2.1 Data Generation and exploration

The data is generated using the `EchoData` class. A call to this class generates `x_batch`, `y_batch`, `x_chunk` and `y_chunk`. `x_batch` contains `batch_size` number of sequences each of length `series_length`. Thus it's shape is (`batch_size`, `series_length`). `y_batch` contains the same data as `x_batch` but it is shifted towards right by `echo_steps` number of steps. It is appended by zeros in the beginning to match the size of `x_batch`. Let us visualize the batches with `batch_size` = 5 and `series_length` = 20000.

```
    x_batch:
0 0 1 0 0 1 0 0 1 0 1 0 0 0 0 1 0 1 1 1 ...
0 1 1 0 0 1 0 1 1 0 1 1 0 1 1 1 0 0 1 1 ...
1 0 0 1 0 0 0 1 1 0 0 1 0 0 1 0 1 1 1 1 ...
1 0 0 0 1 0 1 1 0 1 1 0 0 0 1 1 1 1 1 1 ...
1 0 0 0 1 1 0 1 1 1 0 0 1 0 1 1 0 0 1 1 ...
x_batch size: (5, 20000)

y_batch:
0 0 0 0 0 1 0 0 1 0 0 1 0 1 0 0 0 0 1 0 ...
0 0 0 0 1 1 0 0 1 0 1 1 0 1 1 0 1 1 1 0 ...
0 0 0 1 0 0 1 0 0 0 1 1 0 0 1 0 0 1 0 1 ...
0 0 0 1 0 0 0 1 0 1 1 0 1 1 0 0 0 1 1 1 ...
0 0 0 1 0 0 0 1 1 0 1 1 1 0 0 1 0 1 1 0 ...
y_batch size: (5, 20000)
```

Since backpropagating through a series of length 20000 is practically in feasible, we divide the series in chunks of length `truncated_length`. The RNN is supplied these chunks as inputs sequentially. Let us visualize one of the chunks.

```
    x_chunk:
[0 1 1 0 0 1 0 0 0 0 1 0 1 0 0 1 0 1 1 1]
[0 1 0 0 0 0 1 1 0 0 0 1 1 1 0 1 0 0 0 0]
[1 0 0 0 1 0 0 0 1 0 1 1 1 1 0 1 0 0 0 1]
```

```
[0 0 0 0 0 1 1 0 1 1 1 0 1 0 1 0 0 1 0 1]
[1 1 0 1 0 0 0 1 1 1 0 0 0 1 0 0 1 1 0 1]
1st x_chunk size: (5, 20, 1)

y_chunk:
[1 1 1 0 0 0 0 1 0 1 1 0 0 1 0 0 0 0 0 0]
[0 0 0 1 0 1 1 1 1 1 1 0 1 0 1 0 0 0 0 1]
[0 0 1 1 1 1 1 0 0 0 1 0 1 1 0 1 0 1 0 0]
[1 0 1 0 0 1 0 0 1 0 0 0 0 1 1 1 0 1 0 1]
[1 0 1 1 1 0 0 0 1 1 0 0 0 0 0 0 1 1 0 1]
1st y_chunk size: (5, 20, 1)
```

Note that 1 is appended to the dimension of the chunks because `torch.nn.RNN` assumes the input to be three dimensional where 3^{rd} dimension contains the input dimension which is 1 in our case (since we have binary sequence).

40.2.2 Model Architecture

The following class contains the architecture of the model -

```
class SimpleRNN(nn.Module):
    def __init__(self, input_size, rnn_hidden_size, output_size):
        super().__init__()
        self.rnn_hidden_size = rnn_hidden_size
        self.rnn = torch.nn.RNN(
            input_size=input_size,
            hidden_size=rnn_hidden_size,
            num_layers=1,
            nonlinearity='relu',
            batch_first=True
        )
        self.linear = torch.nn.Linear(
            in_features=rnn_hidden_size,
            out_features=1
        )

    def forward(self, x, hidden):
        x, hidden = self.rnn(x, hidden)
        x = self.linear(x)
        return x, hidden
```

The model architecture is pretty straight forward. The input chunk is inputted in the RNN with 1 hidden layer. `batch_first` set to True which makes the batch size as the first dimension for both input and output. RNN outputs a hidden vector and an output vector. The output vector is of shape (`batch`, `seq_len`, `hidden_size`). Since the output of the model have to be either 0 or 1, a linear layer is used to map the output of RNN from `hidden_size` to a 2 dimensional space.

40.2.3 Training

The model is trained for 5 epochs using the training function shown below.

```
def train(hidden):
    model.train()

    correct = 0
    for batch_idx in range(train_size):
        data, target = train_data[batch_idx]
        data, target = torch.from_numpy(data).float().to(device),
        ↪  torch.from_numpy(target).float().to(device)
        optimizer.zero_grad()
        if hidden is not None: hidden.detach_()
        logits, hidden = model(data, hidden)
        loss = criterion(logits, target)
        loss.backward()
        optimizer.step()

        pred = (torch.sigmoid(logits) > 0.5)
        correct += (pred == target.byte()).int().sum().item()

    return correct, loss.item(), hidden
```

The important thing to note is the use of `hidden.detach_()`. It is necessary to use this as we are reusing the hidden states obtained from one chunk into another chunk. But we want the gradients to be back-propagated only over the sequences in the current chunk and not all the way back. `hidden.detach_()` removes the hidden variable from computation graph and prevents the gradients to back-propagate through the variable.

The loss function used is `BCEWithLogitsLoss`. This loss combines a Sigmoid layer and the BCELoss in one single class where BCELoss measures the Binary Cross Entropy between the target and the output. This is more numerically stable than using a Sigmoid followed by a BCELoss as we take advantage of the log-sum-exp trick for numerical stability.

The model achieves training and test accuracy of 100% after 5 epochs.

40.2.4 Output Visualization

The trained model is used to echo a random sequence. The output is shown below.

```
# Let's try some echoing
my_input = torch.empty(1, 100, 1).random_(2).to(device)
hidden = None
my_out, _ = model(my_input, hidden)
print(my_input.view(1, -1).byte(), (my_out > 0).view(1, -1), sep='\n')
```

```
    tensor([[1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1,
         0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1,
         0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0,
```

```
        0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1,
        0, 1, 1, 0]], dtype=torch.uint8)
tensor([[0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0,
        0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0,
        0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0,
        1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0,
        0, 1, 1, 0]], dtype=torch.uint8)
```

The model successfully echoes the data after 3 steps except the start of the sequence. This is because a None hidden state is supplied to the model and it takes some time to transform into a meaningful state.

Chapter 41

Variational Auto-Encoder

In this chapter, we present the a module to achieve variational auto-encoder(VAE).

41.1 Setup

```
import torch
import torchvision
from torch import nn
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import MNIST
from matplotlib import pyplot as plt

# Displaying routine
def display_images(in_, out, n=1, label=None, count=False):
    for N in range(n):
        if in_ is not None:
            in_pic = in_.data.cpu().view(-1, 28, 28)
            plt.figure(figsize=(18, 4))
            plt.suptitle(label + ' - real test data / reconstructions', color='w', fontsize=16)
            for i in range(4):
                plt.subplot(1,4,i+1)
                plt.imshow(in_pic[i+4*N])
                plt.axis('off')
        out_pic = out.data.cpu().view(-1, 28, 28)
        plt.figure(figsize=(18, 6))
        for i in range(4):
            plt.subplot(1,4,i+1)
            plt.imshow(out_pic[i+4*N])
            plt.axis('off')
            if count: plt.title(str(4 * N + i), color='w')
```

```
# Set random seeds

torch.manual_seed(1)
torch.cuda.manual_seed(1)

# Define data loading step

batch_size = 256

kwargs = {'num_workers': 1, 'pin_memory': True}
train_loader = torch.utils.data.DataLoader(
    MNIST('./data', train=True, download=True,
                  transform=transforms.ToTensor()),
    batch_size=batch_size, shuffle=True, **kwargs)
test_loader = torch.utils.data.DataLoader(
    MNIST('./data', train=False, transform=transforms.ToTensor()),
    batch_size=batch_size, shuffle=True, **kwargs)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

41.2 VAE model

```
d = 20

class VAE(nn.Module):
    def __init__(self):
        super().__init__()

        self.encoder = nn.Sequential(
            nn.Linear(784, d ** 2),
            nn.ReLU(),
            nn.Linear(d ** 2, d * 2)
        )

        self.decoder = nn.Sequential(
            nn.Linear(d, d ** 2),
            nn.ReLU(),
            nn.Linear(d ** 2, 784),
            nn.Sigmoid(),
        )

    def reparameterise(self, mu, logvar):
        if self.training:
            std = logvar.mul(0.5).exp_()
            eps = std.data.new(std.size()).normal_()
            return eps.mul(std).add_(mu)
```

```
        else:
            return mu

    def forward(self, x):
        mu_logvar = self.encoder(x.view(-1, 784)).view(-1, 2, d)
        mu = mu_logvar[:, 0, :]
        logvar = mu_logvar[:, 1, :]
        z = self.reparameterise(mu, logvar)
        return self.decoder(z), mu, logvar

model = VAE().to(device)
```

Here we define our VAE. We define an encoder mapping Linear → RELU → Linear. The output of the encoder is of 2d dimension because we need mean and variance for further sampling. The input in the decoder is of d dimension because for input we used the two parameters returned by encoder to generate, mean and variance and sampled it. The 784 corresponds to the MNIST data dimension (28 * 28). We use log-variance so that we can use the whole range of data instead of only positive numbers using variance. We get our latent variable (z in our case) by multiplying deviation(generated by encoder) to the epsilon(sampled from a normal distribution) and add our mu(generated by encoder). We do not backpropagate through sampling(we only care about variance and mean produced by our encoder, not episilon).

41.3 Training

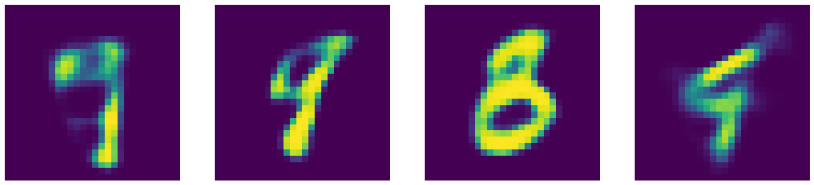
```
learning_rate = 1e-3

optimizer = torch.optim.Adam(
    model.parameters(),
    lr=learning_rate,
)
# Reconstruction + KL divergence losses summed over all elements and batch

def loss_function(x_hat, x, mu, logvar):
    BCE = nn.functional.binary_cross_entropy(
        x_hat, x.view(-1, 784), reduction='sum'
    )
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())

    return BCE + KLD

# Training and testing the VAE

epochs = 11
for epoch in range(1, epochs + 1):
    # Training
    model.train()
```

```
    train_loss = 0
    for x, _ in train_loader:
        x = x.to(device)
        # ===================forward=====================
        x_hat, mu, logvar = model(x)
        loss = loss_function(x_hat, x, mu, logvar)
        train_loss += loss.item()
        # ===================backward====================
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    # ===================log========================
    print(f'====> Epoch: {epoch} Average loss: {train_loss / len(train_loader.dataset):.4f}')

    # Testing

    with torch.no_grad():
        model.eval()
        test_loss = 0
        for x, _ in test_loader:
            x = x.to(device)
            # ===================forward=====================
            x_hat, mu, logvar = model(x)
            test_loss += loss_function(x_hat, x, mu, logvar).item()
    # ===================log========================
    test_loss /= len(test_loader.dataset)
    print(f'====> Test set loss: {test_loss:.4f}')
    display_images(x, x_hat, 1, f'Epoch {epoch}')
```

41.4 Result

41.4.1 Random sample + Decoder

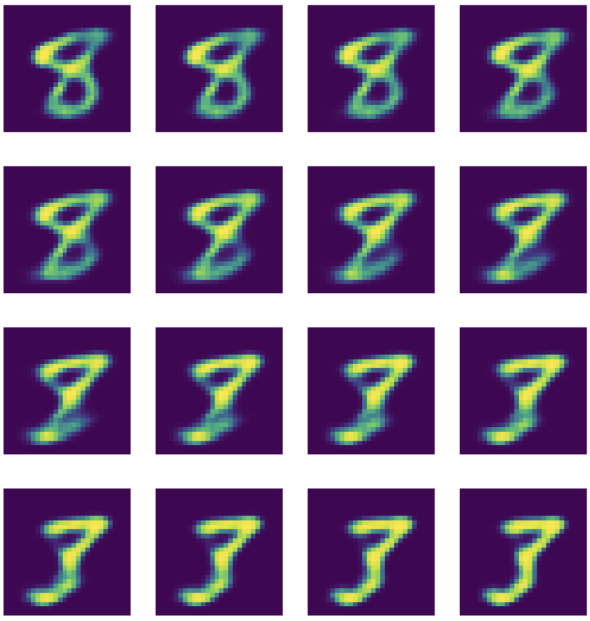
```
# Generating a few samples

N = 16
sample = torch.randn((N, d)).to(device)
sample = model.decoder(sample)
display_images(None, sample, N // 4, count=True)
```



Above is an example of generating something given samples drawn in the latent space with normal distribution instead of using encoder. In the above case 8 was generated starting from random sample in normal distribution. The third picture already looks like an 8.

41.4.2 Smooth interpolation

```
# Perform an interpolation between input A and B, in N steps

N = 16
code = torch.Tensor(N, 20).to(device)
for i in range(N):
    code[i] = i / (N - 1) * mu[B].data + (1 - i / (N - 1) ) * mu[A].data
sample = model.decoder(code)
display_images(None, sample, N // 4, count=True)
```



Above we present a smooth interpolation from an 8 to a 3, in 16 steps.
Linear interpolation in the latent space is warped.

Chapter 42

Bayesian Neural Networks

In this section, we perform experiments using Bayesian neural networks. We show that Bayesian neural networks capture model uncertainty, which cannot be captured by standard neural networks or deep learning tools.

42.1 Libraries

```
import torch
from torch import nn, optim
from matplotlib import pyplot as plt
from plot_lib import set_default
```

42.2 Creating the data

```
# Set style (needs to be in a new cell)
set_default(figsize=(16, 8))

# Training set
m = 20  # nb of training pairs
x = (torch.rand(m) - 0.5) * 10  # inputs, sampled from -5 to +5
y = x * torch.sin(x)  # targets

# View training points
plt.plot(x.numpy(), y.numpy(), 'o')
plt.axis('equal')
plt.ylim([-10, 5])
```

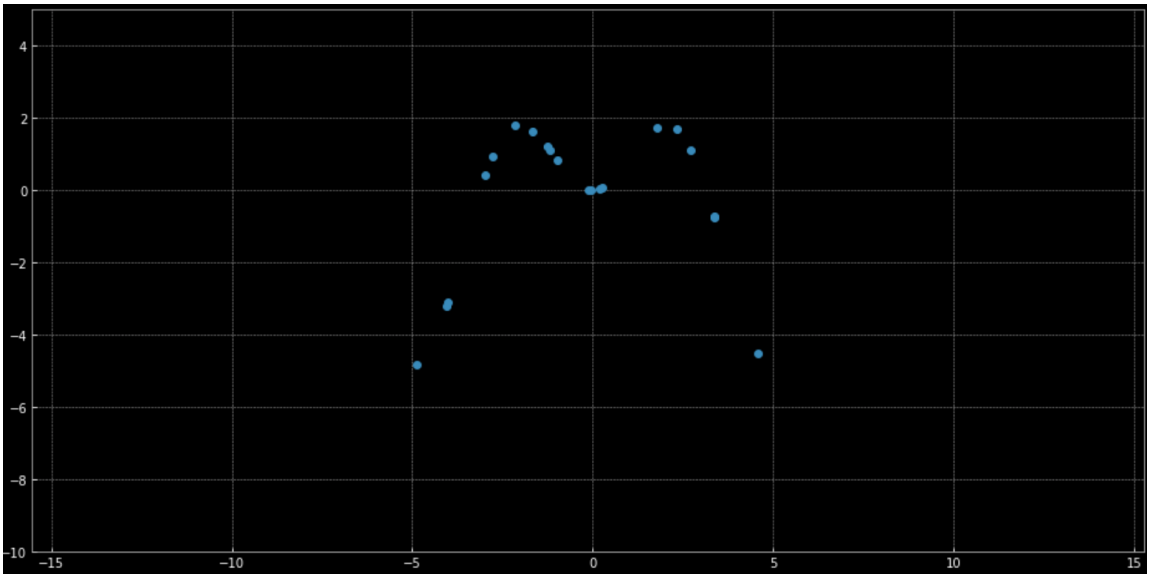


Figure 42.1: View training data

We define 20 training data points from -5 to +5. Now, we are going to fit a network over the data points.

42.3 Experiments using Bayesian neural networks

```
# Define network architecture (try different non-linearities)

non_linear = nn.Tanh
# non_linear = nn.ReLU

net = nn.Sequential(
    nn.Dropout(p=0.05),
    nn.Linear(1, 20),
    non_linear(),
    nn.Dropout(p=0.05),
    nn.Linear(20, 20),
    non_linear(),
    nn.Linear(20, 1)
)
```

In this neural network, the dropout acts directly on our inputs. We use tanh function for non-linearity in our first experiment.

```
# Training objective and optimiser
criterion = nn.MSELoss()
```

```
optimiser = optim.SGD(net.parameters(), lr=0.01, weight_decay=0.00001)

# Training loop
for epoch in range(1000):
    y_hat = net(x.view(-1, 1))
    loss = criterion(y_hat, y.view(-1, 1))
    optimiser.zero_grad()
    loss.backward()
    optimiser.step()
#     print(loss.item())

# Define a denser input range
xx = torch.linspace(-15, 15, 1000)

# Evaluate net over denser input (try both eval() and train() modes)

net.eval()
# net.train()

with torch.no_grad():
    plt.plot(xx.numpy(), net(xx.view(-1, 1)).squeeze().numpy(), 'C1')
plt.plot(x.numpy(), y.numpy(), 'oC0')
plt.axis('equal')
plt.ylim([-10, 5])
```

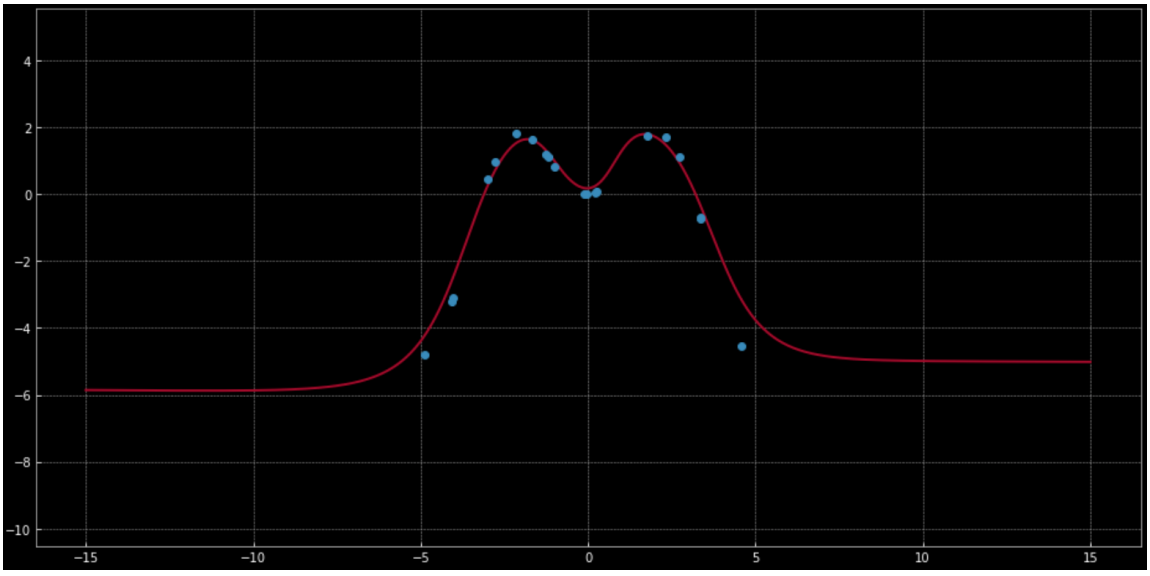


Figure 42.2: Fit standard neural network (with Tanh non-linearity)

We set to evaluation mode to turn-off the dropout. As we can see in Fig. 42.2, it gives us a fine approximation. However, this standard neural network does not show any uncertainty. Standard neural networks only provide point estimates of y values in different locations corresponding to x values. Then, how do we extract uncertainty from the network? First, we start with setting the network to train mode.

```
# Multiple (100) runs for denser input
net.train()
y_hat = list()
with torch.no_grad():
    for t in range(100):
        y_hat.append(net(xx.view(-1, 1)).squeeze())

# Evaluate mean and std over denser input
y_hat = torch.stack(y_hat)
mean = y_hat.mean(0)
std = y_hat.std(0)

# Visualise mean and mean ± std -> confidence range
plt.plot(xx.numpy(), mean.numpy(), 'C1')
plt.fill_between(xx.numpy(), (mean + std).numpy(), (mean - std).numpy(), color='C2')
plt.plot(x.numpy(), y.numpy(), 'oC0')
plt.axis('equal')
plt.ylim([-10, 5])
```

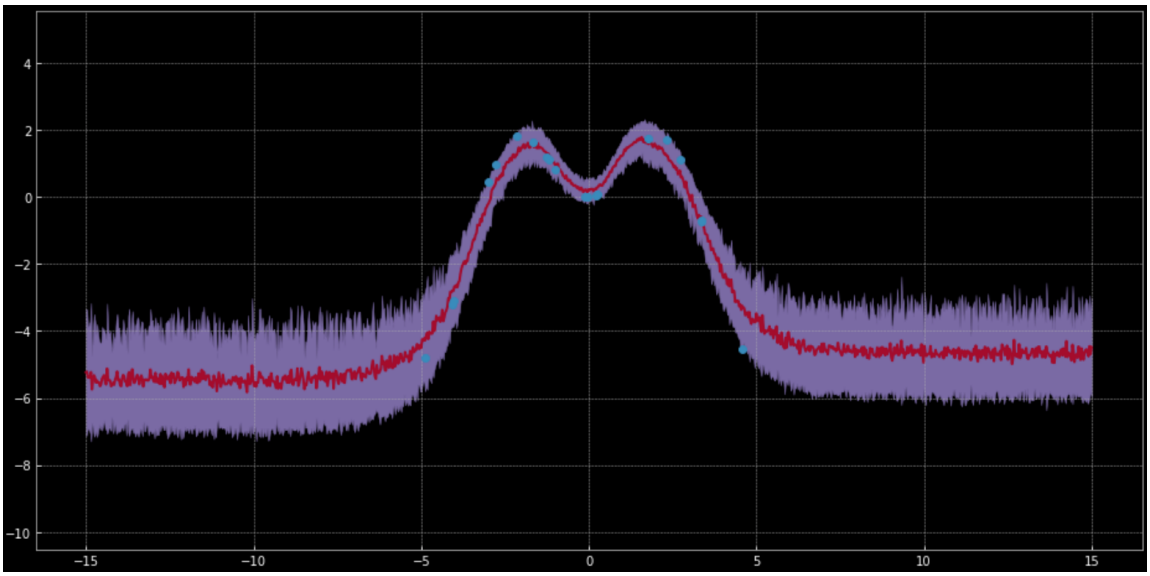


Figure 42.3: Fit Bayesian neural network (with Tanh non-linearity)

Fig. 42.3 provides so much more information compared to the results in Fig. 42.2. The red

curve shows an average of 100 evaluations, and the purple curve is mean plus/minus one standard deviation over the predictions. Thus, this plot not only presents mean estimates but also shows variances that provide uncertainty of the results.

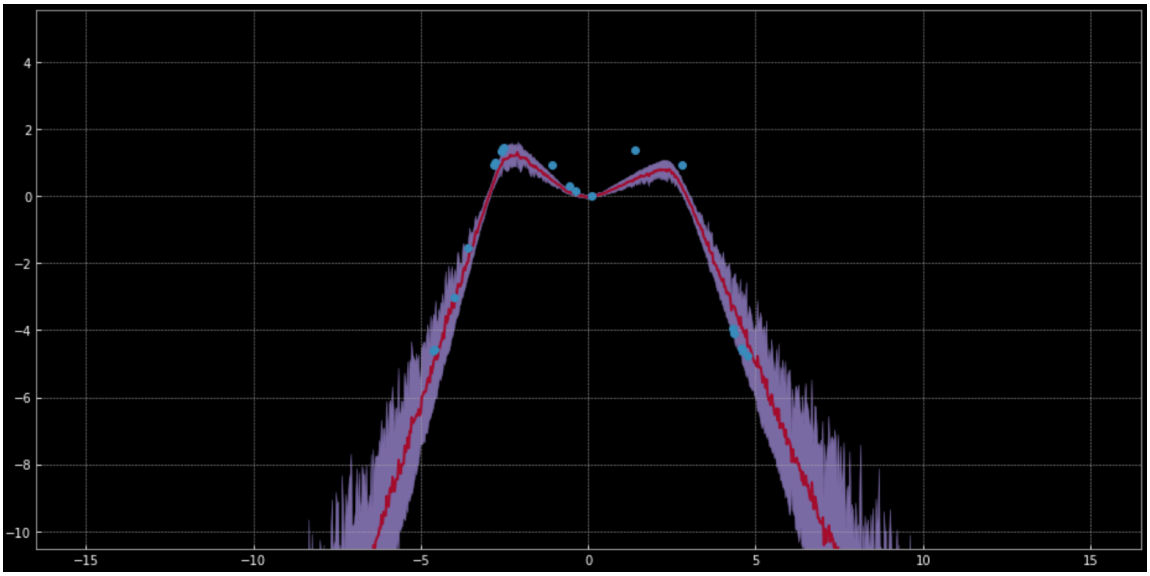


Figure 42.4: Fit Bayesian neural network (with ReLU non-linearity)

Fig. 42.4 presents estimate results using ReLU function. We note that if we use different non-linearities in our network, it provides different uncertainty estimates. It is similar to changing kernels in Gaussian process. Furthermore, the uncertainty (or variance) increases as we go farther from the training points.

Part IV

Applications

Chapter 43

Convolutional Networks Applications in Image Recognition

43.1 Convolution Networks for Scene Parsing and Labeling

Continuing the discussion of Semantic Segmentation we see how one can use convolutional neural networks to categorize and label sections of an image. Using this technique we're able to apply labels to many components of an image. An example of the results can be seen in figure 43.1 below.

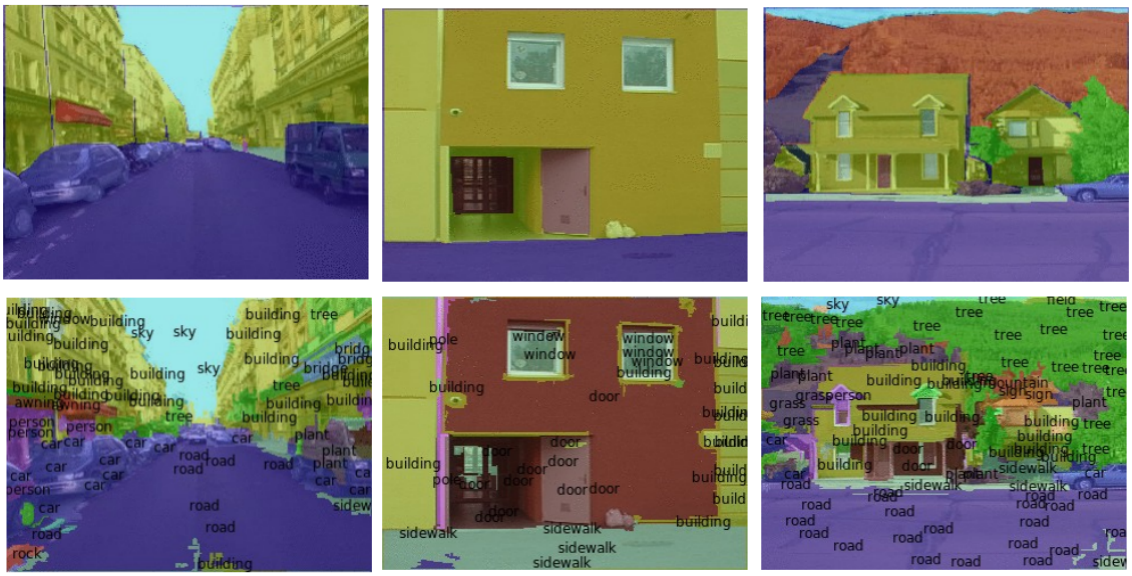


Figure 43.1: Example of images first segmented then labeled from class notes

43.1.1 Multiscale Convolution Neural Network Architecture

In the Multiscale Architecture, as described in Farabet et al. [2013], the model generates multiple versions of the input at different scales. Each version has a kernel applied to it, producing a set of features for each scale. The feature maps are upsampled to the largest size, and categories are predicted on the labeled images in a supervised model. This method allows us to pick out features in the image at a variety of scales.

In order to correctly label an image it must also be segmented. A *superpixel* representation is created by a separate network, which over-segments the image based on local pixel information. Each segment is assigned a category based on the "majority vote" of the features from the multiscaled network. An illustration of the process is shown below in figure 43.1

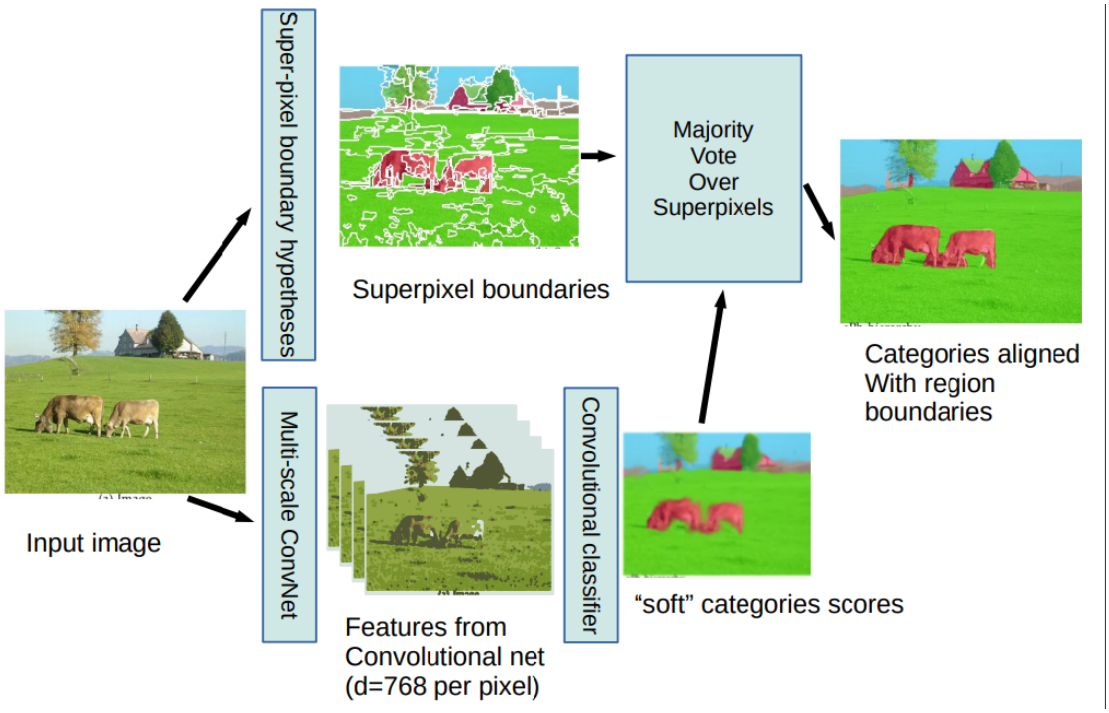


Table 43.1: Flow of the super-pixel/multi-scale system from class notes

43.1.2 RGB+Depth Images

In the article by Couprie et al. [2013] we see how the approach outlined above can be applied to stills from video that captures depth information alongside the RGB image. The depth is treated as a a channel of another color, leaving the model otherwise as it was. The addition of depth is especially beneficial for static objects, like walls, whose depth stays relatively constant across the images.

43.1.3 Performance

A convolutional neural network can perform Scene Parsing of video at a reasonable pace. The network requires no post-processing and can label the images frame-by-frame. For instance, an implementation designed by Farabet et al. [2013] runs at 50ms/frame on Virtex-6 FPGA hardware. Often, the limiting factor in such implementations is network bandwidth, as much is needed to transmit the features. Using the Multiscale approach as described in 43.1.1, a model produced state of the art results at a fraction of the time on the Stanford Background Dataset as seen in table 43.2.

	Pixel Acc.	Class Acc.	CT (sec.)
Gould *et al.* 2009 [15]	76.4%	-	10 to 600s
Munoz *et al.* 2010 [32]	76.9%	66.2%	12s
Tighe *et al.* 2010 [44]	77.5%	-	10 to 300s
Socher *et al.* 2011 [43]	78.1%	-	?
Kumar *et al.* 2010 [25]	79.4%	-	$< 600s$
Lempitzky *et al.* 2011 [30]	**81.9%**	72.4%	$> 60s$
singlescale convnet	66.0 %	56.5 %	0.35s
multiscale convnet	78.8 %	72.4%	0.6s
multiscale net + superpixels	80.4%	74.56%	**0.7s**
multiscale net + gPb + cover	80.4%	75.24%	61s
multiscale net + CRF on gPb	81.4%	**76.0%**	60.5s

Table 43.2: Performance of the system on the Stanford Background dataset

43.2 Deep Convolution Networks for Object Recognition

Traditionally, image recognition has been done with shallow networks over hand-crafted feature-sets trained as a supervised learning problem. As computer power improved and GPU's became more readily available, deeper convolutional networks became more commonplace in image recognition. As the deeper networks were developing, large-scale labeled datasets became more readily available for model training. In 2012, the winning model in the ImageNet Large Scale Visual Recognition Competition, AlexNet by Krizhevsky et al. [2012], had a top-5 error rate of 16.4% compared to the previous years best error rate of 25.8%. With each passing year the winning model has had additional parameters and layers alongside the decreased error rate.

43.2.1 Depth Inflation

As computational power continued to increase and new techniques for training were learned, the depth of networks continued to increase. With VGG from Simonyan et al. [2013] the image is continually downsized at each layer, while the number of features is doubled, allowing it to continue

to learn. Google (Szegedy et al. [2014]) produced GoogleNet, a much deeper model, with parallel layers and multiple side predictions made along the way. ResNet, another model used in the ImageNet competition (He et al. [2015]), had perhaps the largest jump in depth, and it showed that networks with over 150 layers could be successfully trained. ResNet was built with *skip connections* or operations where a layers results were added back to previous results to prevent dying layers that normally stop the learning process. Since then there has been increased complexity in the models, including DenseNet by Huang et al. [2016] which utilizes blocks of skip connections.

43.2.2 Performance Comparisons

There are a series of techniques to evaluate networks on object recognition tasks. The figure below shows the Top-1 Accuracy, the computational complexity by number of operations and the model complexity by number of parameters. We posit that the ideal model is one in the top-left quadrant of the graph as it has both high accuracy and few operations. A model like ResNet-50 and Inception-v3 show some of the reasonable trade-offs needed for a good model. In comparison, ResNet-152 and Inception-v4 have improved accuracy over their brethren at the cost of an increased number of operations. DenseNet, published after the previous graph was produced, achieves a similar accuracy to ResNet-152 at a lower computational cost. DenseNet was further enhanced by utilizing Multiscale technique detailed in 43.1.1 alongside an early stopping condition. Periodically, and before an image has gone through all the layers, a prediction is generated, and if the prediction probability passes a threshold the prediction is made and the operations stopped. The early stopping condition has been shown to operate nearly 2.6 times faster on a similar network without dynamic evaluation.

43.2.3 Future work

In the race to improve performance researchers at Instagram (Mahajan et al. [2018]) have pre-trained models by attempting to predict hashtags from images on an enormous (most recently 3.5B) catalogue of images. As the size of their pre-training set has increased, so has their increase in accuracy. While the paper results are public, this pre-trained model has not been released, instead it is used as a competitive advantage for Instagram/Facebook.

43.3 Statistics

Microsoft estimates the 90% of the deep learning networks running on mobile devices are convolutional neural networks. Hardware and software companies are now building convolutional neural network chips that are directly integrated into smart-phones. In the near future, we should expect cars and children's toys to include these chips as well.

43.4 Facial Recognition

Upon initial upload of photos to Facebook, each photo is sent through several convolutional neural networks where the image is tagged and feature vectors are generated. The initial networks used include networks for identifying images the violate Facebook's guidelines and networks for friend and friend of friend recognition. This process is detailed in Mahajan et al. [2018].

Deep Face

Deepface is an algorithm built for face detection at Facebook and detailed in the following papers, Chopra et al. [2005] and Hadsell et al. [2006]. The algorithm first locates faces in an image then fits the face to a 3D model allowing the face to be unfolded and flattened. The unfolded face is then put through a Siamese network, where the face is then recognized. The siamese network is trained on a multi-part cost function where facial vectors are minimized between the same face and maximized between different faces. Importantly, the algorithm uses Euclidean distance, but is not squared to prevent zeroing out the gradient. The cost functions are defined below.

$$Loss_{sameface} = ||G(x_i) - G(x_j)||$$
$$Loss_{differentfaces} = Max(0, m - ||G(x_i) - G(x_j)||$$

43.5 Object Detection and Localization

43.5.1 Classification + Localization

To train a convolutional neural network for object detection, we train a network on a set input scale, before implementing the network over a series of scales in a sliding window. The model is able to detect objects in the image and the rectangular bounding box bounding the object. More complex models have been used to detect the body pose of a person using these techniques.

R-CNN

Originally proposed in Girshick et al. [2013], the R-CNN uses a selective search algorithm to generate region proposals, and convolutional neural network to generate a proposed region of interest. A secondary classifier is then trained to classify those regions, while a regressor is trained to refine the binding box based on the region proposal location.

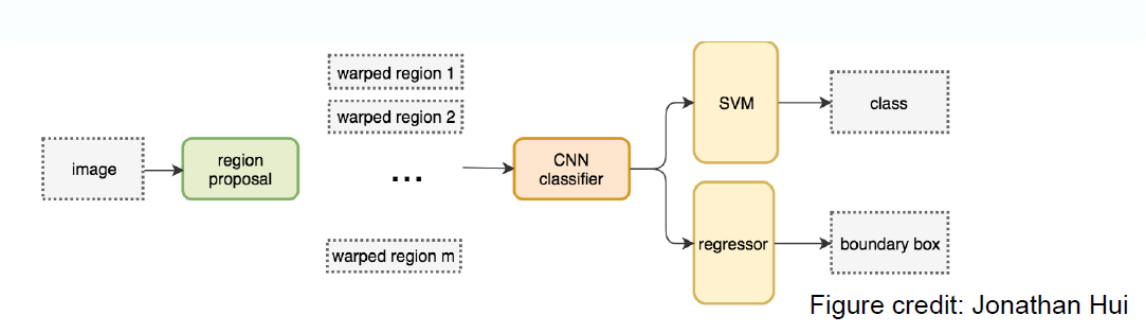


Table 43.3: Architecture of the R-CNN

Fast R-CNN

A more recent evolution of the R-CNN is the Fast R-CNN which feeds the image from the region of interest to the classifier. The regions of interest are extracted from the feature map.

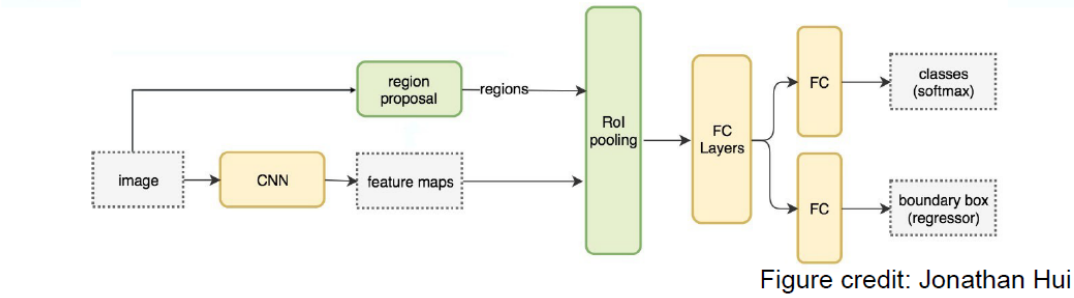


Table 43.4: Architecture of the Fast R-CNN

Faster R-CNN

More recently, the Faster R-CNN has been proposed. The Faster R-CNN differs from the previous Fast R-CNN as the region proposal is achieved by a neural network, instead of the selective search algorithm.

YOLO

YOLO, another convolutional neural network architecture, designed in Redmon et al. [2015] consists of a single network that predicts the bounding boxes and perform the classification for object detection. The network has 24 convolutional layers and 2 fully connect layers. While it does not compare to the R-CNN architecture it may or may not be a faster trained algorithm.

DeepMask: Segmenting and Localizing Objects

DeepMask, proposed in Pinheiro et al. [2015], is a convolutional neural network designed to produce object masks. The region proposal is generated by a discriminative convolutional network followed by a series of convolutions and a classifier. The classifier determines whether a pixel belongs to the object in the centre of a input image patch.

Mask R-CNN

Mask R-CNN, He et al. [2017], uses the architecture of Faster R-CNN to produce a mask for each class. The network adds a branch to predict the object mask in parallel to the bounding box.

RetinaNet

RetinaNet, a feature pyramid network is a convolutional neural network with both down-sampling followed by upsampling with skip connections in between. Initially proposed in Lin et al. [2017], RetinaNet also uses a novel loss function. The cross entropy loss function is utilized alongside a

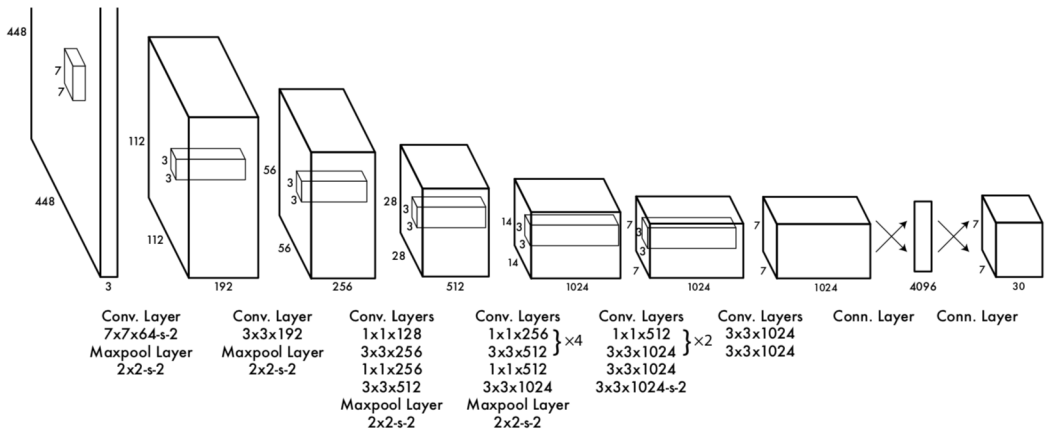


Table 43.5: Architecture of YOLO

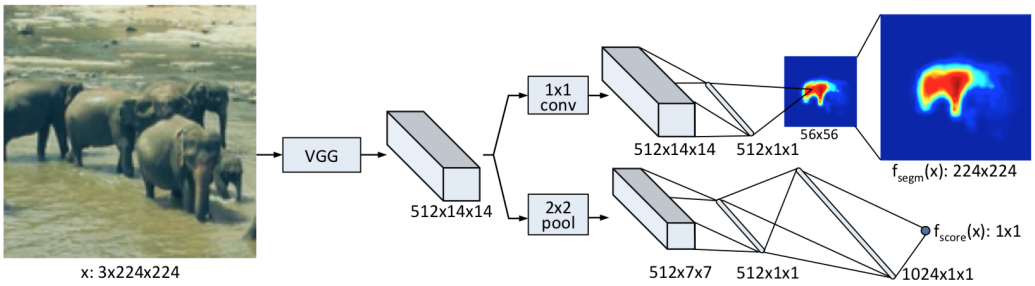


Table 43.6: Architecture of DeepMask

dynamic scaling factor that downweights the easier samples and forces the network to focus on the harder to predict examples during training.

DensePose: Real-Time Body Pose Estimation

DensePose, developed by Facebook this past year in Neverova et al. [2018], learns the surface-based representation of the human body through a Mask R-CNN. The system runs in real-time on a single GPU. During network training, both fully convolutional neural networks and region-based approaches were tested, with the latter proving most effective.

43.5.2 Image Captioning

Image Captioning was first attempted using a convolutional neural network to generate a feature vector which was used as input to an LSTM. By training the LSTM in a supervised manner, the

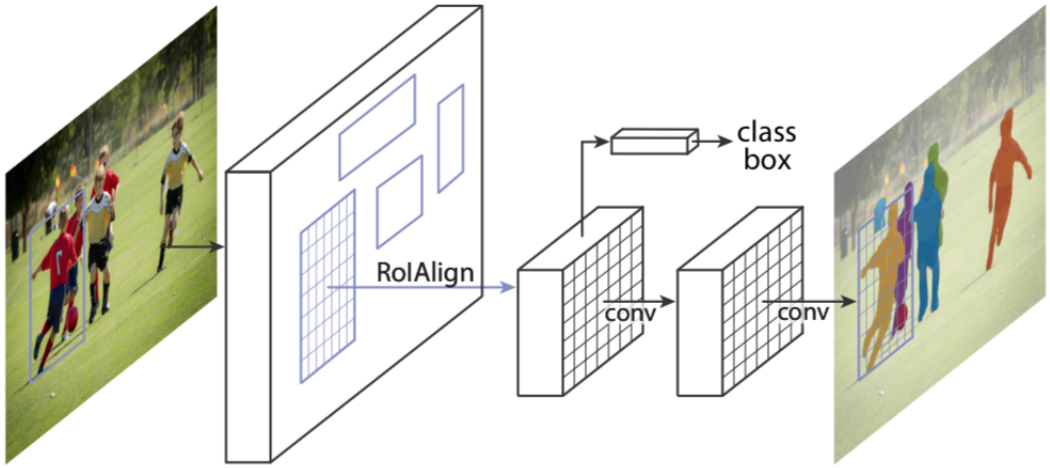


Table 43.7: Architecture of Mask R-CNN

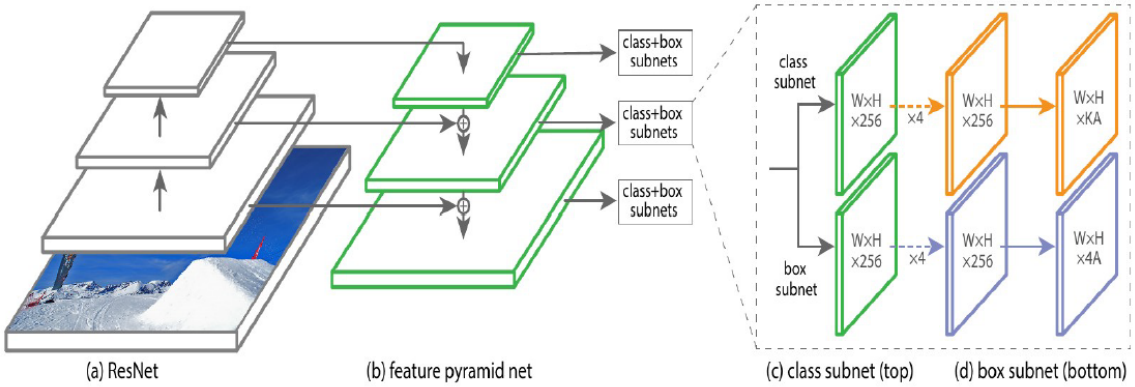


Table 43.8: Architecture of RetinaNet

model can output a caption.

Chapter 44

Applications of Convolutional Networks cont.

44.1 Medical Imaging

CNNs have also been successfully applied to medical imaging. Some examples are:

- Brain Tumor Segmentation Albadawy et al. [2018]
- Skin Cancer Detection Esteva et al. [2017]
- Proximal Femur Segmentation from MR Images Deniz et al. [2018]

However, medical imaging is challenging because of the lack of labeled data.

44.1.1 ConvNets in Connectomics

Connectomics is the study of reconstructing schematics of the brain. Researchers typically proceed by taking a stack of images (thin slices) to get information for the complete volume. One approach to making a 3D reconstruction from 2D images is, of course, using ConvNets Jain et al. [2010]. One challenge of this reconstruction is that the neurons are very densely packed and each shape of the neuron is a strong indicator of its role. Hence, the reconstructions have to be very accurate and high-res.

44.1.2 3D ConvNets for Medical Image Analysis

U-Net

In medical imaging applications, the U-Net architecture is often used. This architecture consists of a cascade of convolutional filters that downsample the input image from high-resolution to low resolution as in a normal ConvNet and then a series of "deconvolution" filters that upsample the representation back to high resolution (along with skip connections across the "U" structure to make up for the loss of resolution during the downsampling phase). Figure 44.1 shows the U-Net architecture. U-Net has won the EM segmentation challenge at ISBI 2012 by a large margin.

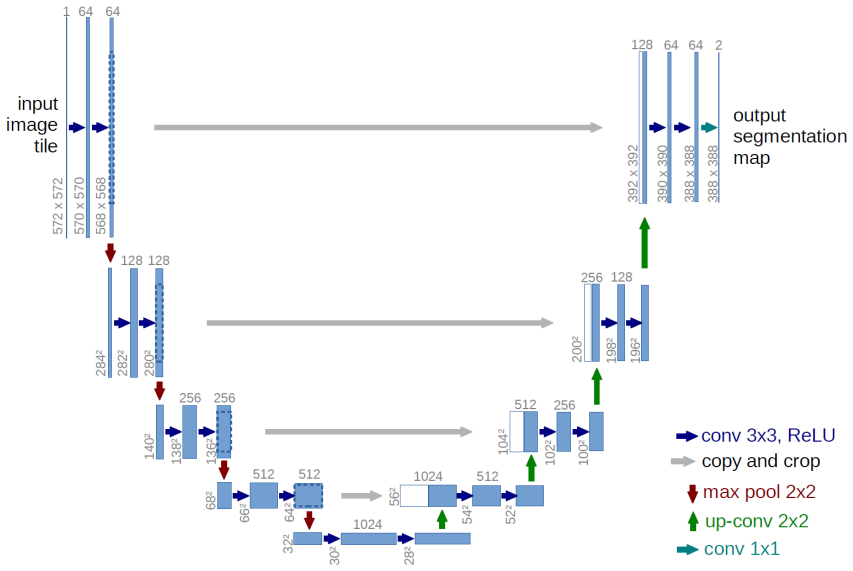


Figure 44.1: U-net architecture from [Ronneberger et al., 2015]

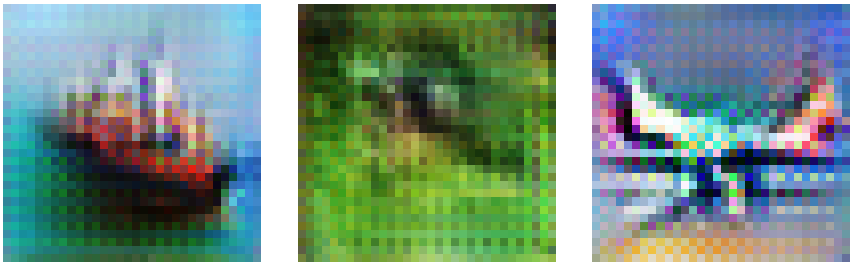


Figure 44.2: checkerboard artifacts from [Odena et al., 2016]

However, one issue with deconvolution is that it can produce "checkerboard" artifacts because of uneven overlaps. Deconvolution is essentially a convolution in reverse, except the pooling step now is replaced with upsampling. There are several different ways to do this upsampling. One naive choice is just repeating the same value over all pixels in the upsampled region, which can lead to aforementioned checkerboard artifacts. Another choice is bilinear interpolation between nearby pixels at the lower resolution to get smoother upsampled image. A third way is a switch type upsampling that tries to mimic an inverse to max-pooling. This can be implemented by remembering the pixel position selected by max pooing during the downsampling step and, in the upsampling step, copying the low-res pixel to the corresponding upsampled pixel position and filling in other pixels with 0's.

44.2 Point Cloud Data

Some 3D data can be represented as 2D images along with estimates of the depth of each pixel. This depth estimate comes from either stereo cameras or LIDAR systems. Unfortunately, the LIDAR is very expensive and often unreliable (eg. doesn't work well when it rains) so some applications like self-driving cars are trying to find other ways to estimate depth.

Point cloud (Figure 44.3) is another way of representing 3D data. The data here consists of a "cloud" of (x, y, z) coordinates along with RGB values. However, point clouds are very sparse since most voxels do not contain any object. This causes problems for traditional ConvNets because the input would be almost entirely zeros.

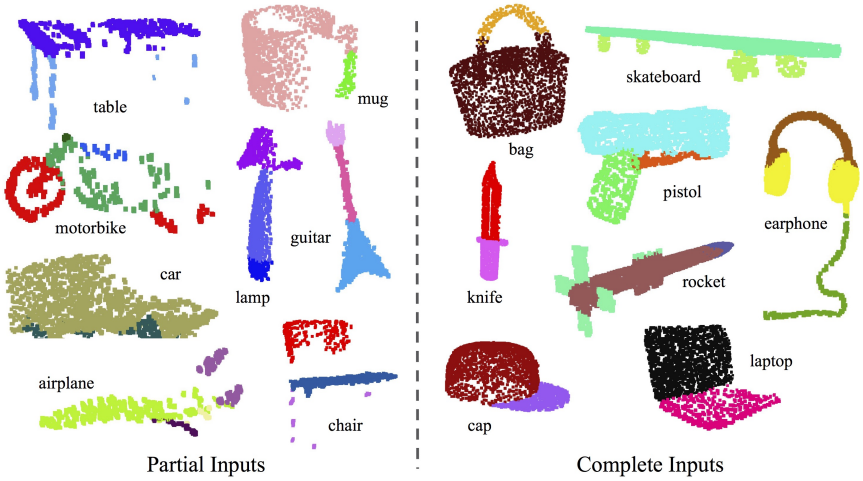


Figure 44.3: Point cloud images from [Qi et al., 2017]

Different variants of ConvNets have been invented to work with this kind of data. For example, there was a ShapeNet competition on semantic segmentation of 3D point cloud data. The winner was the "Submanifold Sparse ConvNet". This method implements *sparse convolutions* that efficiently operate on sparse data. If we use normal convolutions on sparse inputs, then a lot of the computation is wasted since we would be multiplying zeros most of the time. The *sparse convolutions* introduced in this work prevent this waste by keeping track of which "sites" contain information (via a hash table) and not performing convolutional operations on those sites. Another alternative is the various variants of graph CNNs which extend ConvNets to data with non-gridlike structure.

44.3 Speech Recognition

ConvNets can also be used on audio data. To feed audio data to a ConvNet, we can use spectrograms or cepstrograms, which are 2-dimensional plots that show changes in the frequency component of the audio signal with respect to time.

There are mainly two approaches to speech recognition:

1. Classical approach: First train an HMM/GMM model to align the phones with the audio signal, and then train the model to map audio signals to these aligned phones.

2. End-to-end approach (Wav2Letter): The model goes from speech signal to letter transcription directly without intermediate phonetic transcriptions. The input data can be either a spectrogram or even raw waveform Collobert et al. [2017].

The aim of an Automatic Speech Recognition(ASR) system is to find the most likely sentence (word sequence) W which transcribes the speech audio A, with acoustic model $P(A|W)$ and language model $P(W)$. The general process of utilizing ConvNets for speech recognition can be illustrated as Figure 44.4. Useful techniques for speech feature extraction are Short-time Fourier Transforms(STFTS), Mel-frequency cepstrum (MFFCs), Linear Predictive Coding (LPC), etc (44.5). Then a ConvNet is trained on labeled transcriptions for frame-level classification. Further alighment techniques such as Hidden Markov Models will be applied to get phones. Then the alighed phones will be fit into some pre-trained lexicon model to get the corresponding words. At last, a language model will be applied to the words to get the transcription.

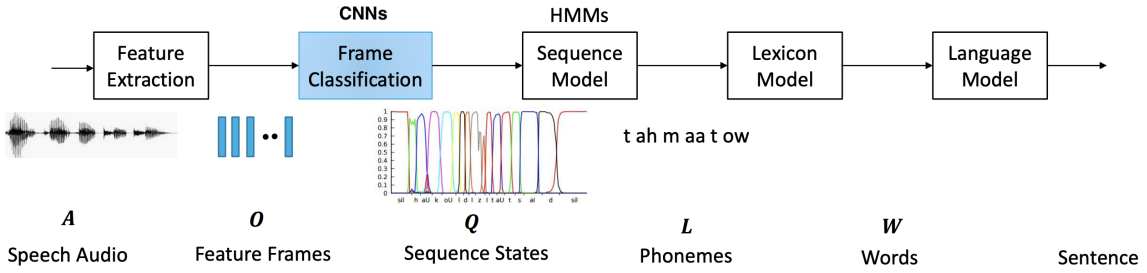


Figure 44.4: CNN-HMM ASR system architecture

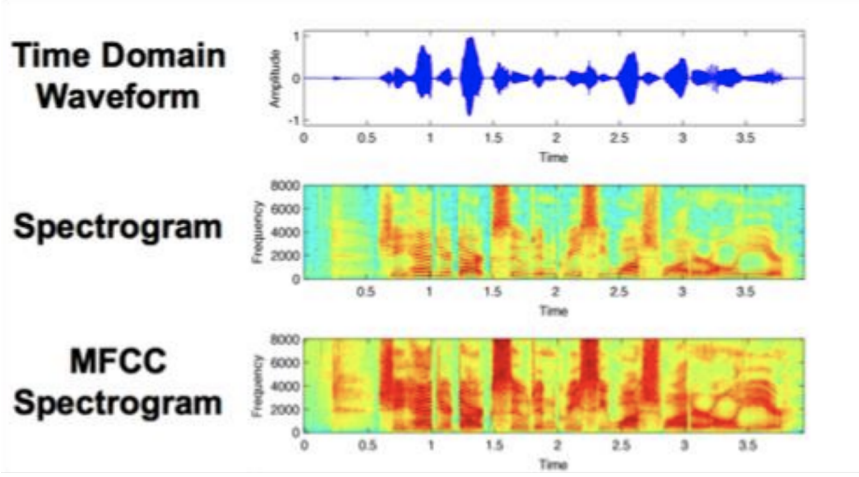


Figure 44.5: speech audio representations

Part V

Papers summary

The end.

Bibliography

Ehab Albadawy, Ashirbani Saha, and Maciej Mazurowski. Deep learning for segmentation of brain tumors: Impact of cross-institutional training and testing. *Medical Physics*, 45, 01 2018. doi: 10.1002/mp.12752.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.

Sumit Chopra, Raia Hadsell, and Yann LeCun. Learning a similarity metric discriminatively, with application to face verification. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, CVPR '05, pages 539–546, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2372-2. doi: 10.1109/CVPR.2005.202. URL http://dx.doi.org/10.1109/CVPR.2005.202.

Ronan Collobert, Christian Puhrsch, and Gabriel Synnaeve. Wav2letter: an end-to-end convnet-based speech recognition system. *CoRR*, abs/1609.03193, 2017.

Camille Couprie, Clément Farabet, Laurent Najman, and Yann LeCun. Indoor semantic segmentation using depth information. *CoRR*, abs/1301.3572, 2013. URL http://arxiv.org/abs/1301.3572.

Cem M. Deniz, Spencer Hallyburton, Arakua Welbeck, Stephen Honig, Kyunghyun Cho, and Gregory Chang. Segmentation of the proximal femur from mr images using deep convolutional neural networks. In *Scientific Reports*, 2018.

Andre Esteva, Brett Kuprel, Roberto A. Novoa, Justin Ko, Susan M. Swetter, Helen M. Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542:115–118, 2017.

Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. Learning hierarchical features for scene labeling. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1915–1929, August 2013. ISSN 0162-8828. doi: 10.1109/TPAMI.2012.231. URL http://dx.doi.org/10.1109/TPAMI.2012.231.

Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.

Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013. URL http://arxiv.org/abs/1311.2524.

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, 2010.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *Proc. Computer Vision and Pattern Recognition Conference (CVPR'06)*. IEEE Press, 2006. Video.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017. URL http://arxiv.org/abs/1703.06870.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9: 1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016. URL http://arxiv.org/abs/1608.06993.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

Viren Jain, H Sebastian Seung, and Srinivas C Turaga. Machines that learn to segment images: a crucial technology for connectomics. *Current Opinion in Neurobiology*, 20(5): 653 – 666, 2010. ISSN 0959-4388. doi: https://doi.org/10.1016/j.conb.2010.07.004. URL http://www.sciencedirect.com/science/article/pii/S0959438810001121. Neuronal and glial cell biology – New technologies.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999134.2999257.

Prechelt L. *Early Stopping - But When?*, volume 1524. Springer, Berlin, Heidelberg, 1998.

Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. *CoRR*, abs/1708.02002, 2017. URL http://arxiv.org/abs/1708.02002.

Dhruv Mahajan, Ross B. Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens van der Maaten. Exploring the limits of weakly supervised pretraining. *CoRR*, abs/1805.00932, 2018. URL http://arxiv.org/abs/1805.00932.

Natalia Neverova, Riza Alp Güler, and Iasonas Kokkinos. Dense pose transfer. *CoRR*, abs/1809.01995, 2018. URL http://arxiv.org/abs/1809.01995.

Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. *Distill*, 2016. doi: 10.23915/distill.00003. URL http://distill.pub/2016/deconv-checkerboard.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, 2012.

Pedro H. O. Pinheiro, Ronan Collobert, and Piotr Dollár. Learning to segment object candidates. *CoRR*, abs/1506.06204, 2015. URL http://arxiv.org/abs/1506.06204.

Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 2017.

Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015. URL http://arxiv.org/abs/1506.02640.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing. ISBN 978-3-319-24574-4.

Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, abs/1312.6034, 2013. URL http://arxiv.org/abs/1312.6034.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. URL http://arxiv.org/abs/1409.4842.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: Lessons learned from the 2015 MSCOCO image captioning challenge. *CoRR*, abs/1609.06647, 2016. URL http://arxiv.org/abs/1609.06647.