Applied Causal Inference Powered by ML and AI

Victor Chernozhukov* Christian Hansen† Nathan Kallus‡
Martin Spindler§ Vasilis Syrgkanis¶

March 5, 2025

Publisher: Online
Version 0.1.1

* MIT
† Chicago Booth
‡ Cornell University
§ Hamburg University
¶ Stanford University

Feature Engineering for Causal and Predictive Inference 10

"It's all about paying attention. [...] Attention is vitality. It connects you with others."

– Susan Sontag [1].

10.1 Introduction 269
10.2 From Principal Components to Autoencoders 270
Variational Autoencoders 274
10.3 From Autoencoders to General Embeddings 275
10.4 Text Embeddings . . . 276
Revisiting the Price Elasticity for Toy Cars 287
10.5 Image Embeddings . . 288
10.6 Application: Hedonic Prices 289
10.7 Notes 293
10.8 Notebooks 293
10.9 Exercises 293

Here we discuss feature engineering as an approach to transform complex objects such as text and images into a collection of relatively low-dimensional numerical features (embeddings) that can be used for standard predictive or causal applications, for example as regressors in a prediction problem. We consider principal components, autoencoders and neural networks as general approaches to generate embeddings. We then consider text embeddings in detail, introducing two popular neural network-based Natural Language Processing (NLP) algorithms: ELMo and BERT. We finally consider image embeddings, applying a hedonic price model to apparel data using a neural network algorithm (ResNet50) to generate embeddings.

10.1 Introduction

Thus far, we have imposed a significant restriction on the kinds of data on which we can perform inference. While empiricists often consider simple datasets that include variables that have a numeric representation (binary, factor and continuous variables), researchers are increasingly confronted with complex forms of data, such as images and text, that encode a vast amount of information. In this section, we generalize our approach to allow using these types of data.

As a motivating example, we consider the problem of predicting prices of products using the types of characteristics that one might find on a webpage, namely the text in the product description and the product's image. The resulting predicted prices are called hedonic prices, and predictive modeling of this form is motivated by the hedonic price models of economics.

In order to predict prices, we have to convert text and images into relatively low-dimensional numerical features, called *embeddings* or *encodings*. The minimal requirement on embeddings is that similar products should have similar embeddings. This requirement guarantees that price predictions for similar products are also similar. The maximal requirement on embeddings is that they should parsimoniously approximate as much information as possible from text and images that is relevant for price predictions.

The main methods for generating successful embeddings include the following, in order of increasing generality:

- classical principal component analysis,
- autoencoders, and
- neural networks solving auxiliary prediction tasks.

The auxiliary tasks in the final method may include solving image processing problems, such as object classification and image compression, or natural language processing problems, such as summarization and machine translation.

These auxiliary tasks are not the same as the "main" task. In our price prediction example, the main task is predicting product prices. Before turning to the primary price prediction task, we might consider running our image and text data through neural networks designed to perform well on image classification or natural language processing. We can then extract features of these networks – embeddings – to use as summaries of the image and text data that are useful for predicting the image type and semantic context of the text. For example, we could

For example, one might use ResNet 50, a pretrained residual network of depth 50, which performs well on image classification tasks and/or a language processing neural network such as BERT. We will discuss such neural networks later in this chapter.

take the final hidden layer of neurons in the simple deep neural networks discussed in Chapter 8 as our embeddings as these neurons are the constructed features that are used in forming the final prediction for the outcome of interest – image type or a language processing task. These embeddings then provide useful inputs for solving the auxiliary object classification or text description task. Since product type and product description are likely relevant determinants of price, these embeddings produced by the auxiliary tasks can serve as useful inputs to the main task – price prediction.

Embeddings are useful in a variety of predictive and causal inference problems. For example, we can imagine using

- embeddings of product images and descriptions for modeling variety and demand for products,
- embeddings of text resumes for studying the wage offer structure,
- or embeddings of countries' characteristics for studying the effect of institutions.

There is an emerging literature on the use of embeddings for causal inference; see this repository of papers about using text data in causal inference. See also [2] for a recent review article on the importance and subtleties of using text as data in the social sciences.

https://github.com/causaltext/causal-text-papers

10.2 From Principal Components to Autoencoders

Principal components are an early classical example of embeddings. One way to frame principal components is that principal components find unit length orthogonal linear combinations, directions, of a collection of variables that are "best" at reproducing the underlying data. The idea is then that a small number of principal components should capture most of the variability in the original variables and thus may provide a useful low-dimensional summary of the original data.

Specifically, let $(W_1, \ldots, W_n)$ be a sample of n observations of a high-dimensional centered[1] random vector W in $\mathbb{R}^d$, and let $\Sigma_n = \mathbb{E}_n[WW'] \in \mathbb{R}^{d\times d}$ denote the empirical covariance matrix. In order to reduce the dimension of W, suppose we wish to find $K \ll d$ mutually orthogonal rotations

1: Thus, $\mathbb{E}_n[W_j] = 0$ for $j = 1, \ldots, d$.

$$X_{ki} := c_k' W_i, \quad k = 1, \ldots, K,$$

of the original W_i's where

$$c'_\ell c_k = 0 \text{ for } \ell \neq k \text{ and } c'_k c_k = 1 \text{ for each } k$$

such that linear combinations of these variables approximate the original data. These rotations are called principal components of W_i. In applications, W_i represent high-dimensional raw features (images, for example), and the principal components

$$X_i^K = (X_{i1}, \ldots X_{iK})'$$

represent a lower-dimensional encoding or embedding of W_i.

More formally, we wish to solve

$$\min_{\{a_j\}_{j=1}^d, \{c_k\}_{k=1}^K} \sum_{j=1}^d \sum_{i=1}^n (W_{ji} - \hat{W}_{ji})^2$$

subject to

$$\begin{aligned} \hat{W}_{ji} := a'_j X_i^K \text{ for } X_i^K = (X_{1i}, \ldots X_{Ki})', \ j = 1, \ldots, d, \text{ and } i = 1, \ldots, n; \\ X_{ki} = c'_k W_i \text{ for } i = 1, \ldots, n \text{ and } k = 1, \ldots, K; \\ c'_k c_k = 1 \text{ for } k = 1, \ldots, K; \\ c'_k c_\ell = 0 \text{ for } \ell \neq k. \end{aligned}$$

The constructed variables resulting from solving this problem,

$$X_i^K = (X_{1i}, \ldots X_{Ki})'$$

are the first K principal components.

Remark 10.2.1 The analytical solution to the principal components problem is as follows: The optimal $C_K = [c_1, \ldots, c_K]$ are the eigenvectors of $\Sigma_n = \mathbb{E}_n[WW']$ corresponding to the K largest eigenvalues $\lambda_1, \ldots, \lambda_K$ of Σ_n. That is, $\Sigma_n c_k = \lambda_k c_k$ for each k. Furthermore, the optimal a_j is the j-th column of C'_K.

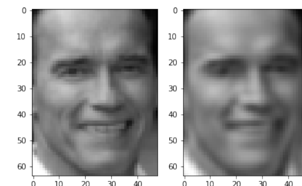

Figure 10.1: Featurizing a talented man: The original 3072-dimensional image W and image $\hat{W}$ produced from a 256-dimensional principal component embedding. As a by-product, we've just made an important causal discovery that, surprisingly, doing embedding causes one to be younger ;).

Another interesting feature of principal components is that they satisfy

$$\mathbb{E}_n[X_k^2] = \lambda_k$$

for $k = 1, \ldots, K$ and

$$\mathbb{E}_n[X_k X_\ell] = 0$$

for $\ell \neq k$. These properties result from the fact that the c_k are eigenvectors of Σ_n.

Finding principal components offers one way to produce em-

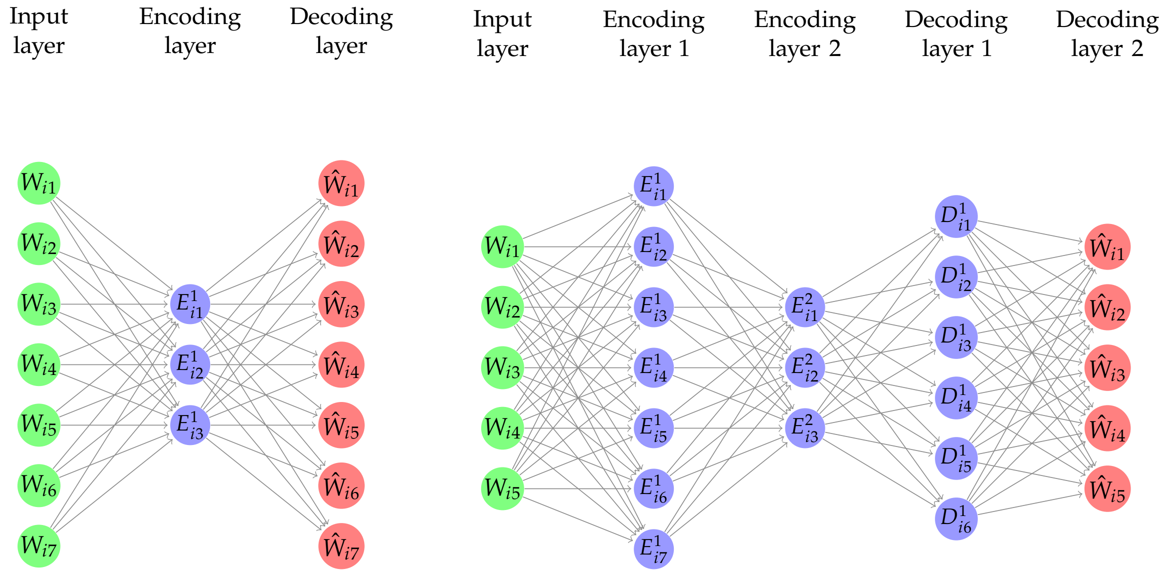

Figure 10.2: The left panel shows a linear single layer autoencoder, such as linear principal components. The right panel shows a three layer nonlinear autoencoder; the middle layers can be used as embeddings.

beddings of raw inputs. Once we have embeddings from any method, we can look at how similar the raw inputs W_k and W_l are via the cosine similarity of the embeddings:

$$\text{sim}(W_k, W_l) = X_k' X_l / (\|X_k\| \|X_l\|).$$

In the context of product embeddings, this approach can be used, for example, to find products that are similar to a given product.

The predictive exercise underlying principal components can be seen as a linear neural network:

$$\underset{d\times 1}{W_i} \longmapsto C_K' W_i =: \underset{k\times 1}{E} \longmapsto A'E =: \underset{d\times 1}{\hat{W}_i},$$

for $A = [a_1, ...a_d]$. The first step is said to be "encoding" the information in the input, and the second step is said to be "decoding" in the sense of returning the encoded information to the original space. Therefore, principal components are embeddings generated by a linear "encoder-decoder" network (an *autoencoder*, for short). For principal components, the relationship between the encoder and decoder happens to be rather simple, in that $A = C_K'$ (see Remark 10.2.1).

This framing suggests that we can immediately generalize this approach to nonlinearly generated encoders and decoders that

have multiple layers:

$$W_i \overset{g_1}{\longmapsto} E_i^1 \ldots \overset{g_k}{\longmapsto} E_i^k \overset{g_{k+1}}{\longmapsto} D_i^{k+1} \ldots \overset{g_m}{\longmapsto} D_i^m =: \hat{W}_i,$$

where maps g_ℓ's are neuron-generating maps. The middle layer or layers of low dimension, represented by the E_i^k, are taken to be encoders. The layers of neurons are mnemonically labelled as either "E" or "D," depending on whether they are doing "encoding" or "decoding," though note that there is no strict formal distinction between these types of layers.

Autoencoders are a way of discovering latent, low-dimensional structures in a dataset. In particular, a random data vector $W \in \mathbb{R}^d$ can be said to have low-dimensional structure if we can find some "well-behaved" functions $\mathsf{e} : \mathbb{R}^d \to \mathbb{R}^k$ and $\mathsf{d} : \mathbb{R}^k \to \mathbb{R}^d$, with $k \ll d$, such that

$$(\mathsf{d}(\mathsf{e}(W)) \approx W.$$

In other words, $X = \mathsf{e}(W)$ is a parsimonious, k-dimensional representation of W that contains all of the information necessary to approximately reconstruct the full vector W. Traditionally, the map $\mathsf{e}(\cdot)$ is called an encoder, and the map $\mathsf{d}(\cdot)$ is called a decoder function. Given this, a general formulation of autoencoders is to minimize the average reconstruction loss,

$$\mathbb{E}_n[\text{loss}(W, \mathsf{d}(\mathsf{e}(W))],$$

over "well-behaved" functions $\mathsf{d} \in \mathsf{D}$ and $\mathsf{e} \in \mathsf{E}$. These classes are often linear, as in principal components, or generated via neural networks.

The qualification of "well-behaved" is important since it is always possible to write down some (completely wild) one-to-one function $\mathsf{e} : \mathbb{R}^d \to \mathbb{R}^1$ such that $\mathsf{e}^{-1}\mathsf{e}(W) = W$.[2]

2: Google "Borel Isomorphism."

Remark 10.2.2 (Independent Component Analysis) Principal component analysis defines "well-behaved" functions as linear functions whose output $(X_1, \ldots, X_k) = e(W)$ has uncorrelated entries, i.e. $\mathrm{E}[X_i X_j] = 0$. In other words, PCA tries to find latent embedding vectors $(X_1, \ldots, X_k)$ that have the ability to reconstruct the original covariate vectors and are uncorrelated with each other. One could take a step further and require that these latent embeddings are independent of

each other, not just uncorrelated, i.e. $X_i \perp\!\!\!\perp X_j$. This leads to the method called linear Independent Component Analysis (ICA) [3]. The intuition of ICA when taken to neural network representations has led to the notion of disentangled representations, i.e. embeddings that encode independent latent dimensions of variation in the data. However, beyond the linear ICA setting, non-linear neural network based versions of ICA have more brittle theoretical foundations in the absence of auxiliary task-related information and task-related outcome variables [4].

Variational Autoencoders

The encoding and decoding functions so far in our discussion have been restricted to be deterministic. Implicitly, this assumes that given the observed high-dimensional variables W, we can uniquely identify the low-dimensional variables that contain all the information in W. Such unique mapping from observed factors to latent factors is not always possible.

An important extension of the autoencoder framework is allowing for these mappings to be stochastic. This extension is inspired by Bayesian probabilistic modelling that views the embeddings as latent factors and imagines that the data are drawn by first drawing the latent factors and then drawing the data samples from some distribution that is dependent on the latent factors. The variational autoencoder [5] attempts to reverse engineer this problem and learn the latent factor space and the posterior distribution of the latent factors conditional on the observed data using computationally tractable approximate versions of the maximum likelihood method.

Although arrived at via different reasoning, variational autoencoders ultimately look similar to autoencoders, albeit introducing randomness in the encoding phase. Roughly speaking, variational autoencoders optimize a loss of the form:

$$\mathbb{E}_n[\text{loss}(W, \mathsf{d}(\mathsf{e}(W, Z))] + penalty(\mathsf{e}),$$

where Z is an exogenous jointly independent Gaussian random noise vector and the penalty term forces the encoding function to be non-deterministic and stems from derivations related to the objective of learning the posterior distribution of latent factors. Conditional on an observed W, the random variable $\mathsf{e}(W, Z) \mid W$ can be interpreted as a random sample from the posterior distribution of the latent factors that could have generated the observed sample W. Moreover, the function

$e(W, Z)$, is typically of the form $e(W, Z) = \mu(W) + \Sigma(W)Z$, where the deterministic functions $\mu(W)$ and $\Sigma(W)$ encode the mean and the covariance of the posterior distribution of the latent factors. These deterministic functions $\mu(W), \Sigma(W)$ can be viewed as deterministic embeddings of W and can be used as engineered features in downstream tasks; see e.g. [6]. Alternatively, one can use only $\mu(W)$, which approximates the posterior mode, as the embeddings. For a more in-depth introduction to variational autoencoders, see [7].

10.3 From Autoencoders to General Embeddings

We can generalize from autoencoders by considering other loss functions where the target outcome in the loss, A, need not be W. Within this context, we can still search for embeddings that minimize average prediction loss for target A,

$$\mathbb{E}_n[\text{loss}(A, \mathsf{f}(\mathsf{e}(W))],$$

where the role of $\mathsf{f}(\cdot)$ is no longer just to decode (i.e. predict W) but rather to predict A.

For example, in feature engineering from images, A could be a product type or subtype, and W could be the image. In feature engineering from text, A could be a masked word in a sentence and W the sentence containing this word. These alternative approaches could be more useful in relation to the final learning task. For example, to build good hedonic price models, we may be much more interested in image or text embeddings that best help to accurately describe the type or subtype of a product than we are in embeddings that are useful for reconstructing the entire image or text itself.

Learning general embeddings with a generic target A is generally implemented via neural networks with structure

$$W_i \overset{g_1}{\longmapsto} E_i^1 \dots \overset{g_k}{\longmapsto} E_i^k \overset{g_{k+1}}{\longmapsto} F_i^{k+1} \dots \overset{g_m}{\longmapsto} F_i^m =: \hat{A}_i.$$

In this structure, g_ℓ's are neuron-generating maps. The middle layers E_i^k are taken to be embedding layers. The F_i^k's are predictive layers which are meant to create good predictions of auxiliary targets.

10.4 Text Embeddings

First generation: Word2Vec Embeddings

We first review some basic ideas underlying the Word2Vec algorithm [8]. One way we could encode words that appear in a corpus of documents (e.g. product descriptions) into a vector is to consider a very high-dimensional vector of dimension d, where d is the total number of words in the corpus. Then the j-th word in the corpus (e.g. in alphabetical order) can be represented as:

$$e_j = (0, ...0, 1, 0, ...0)',$$

with 1 in the j-th position. This encoding has a very high dimension limiting its usefulness. Furthermore, this representation does not capture word similarity – e.g., cosine similarity between two different words j and k is always zero since $e_j{}'e_k = 0$.

Instead we aim to represent words by vectors of much lower dimension, r, that are able to capture word similarity. We denote the representation of the j-th word by u_j, so the dictionary is an $r \times d$ matrix

$$\omega = \{u_1, ..., u_d\},$$

where r is the reduced dimensionality of the dictionary. This dictionary is a linear rotation of the original dictionary $E = \{e_1, ..., e_d\}$, where

$$\omega = \omega E.$$

Therefore, the problem of finding the rotation ω is analogous to the problem of finding principal components, except that our goal is now to find representations ω that are able to capture word similarity. Once we are done, each word t_j in a human-readable dictionary can be represented by a new "word" u_j. The goal of Word2Vec is to find an effective representation with the dimension r of the embedding being much smaller than the total number of words in the corpus, d. We achieve this goal by treating ω as parameters and estimating them so that the model performs well in some basic natural language processing tasks. These tasks are typically not related to downstream tasks, such as predicting hedonic prices or performing causal inference using text as control features, but are related to language prediction tasks.

Figure 10.3 shows components of embeddings for several words produced by a trained Word2Vec map. The numbers presented in the table are not particularly interpretable in isolation. Each

Example of Word2Vec Features

womens	0.388	0.031	-0.197	0.180	-0.223	-0.607	0.306	-0.597
mens	0.759	0.372	0.370	0.707	-0.125	0.509	0.106	0.209
clothing	0.149	0.516	-0.028	0.218	-0.851	-0.410	0.386	0.171
shoes	1.324	-0.359	-0.008	-0.552	0.011	0.365	0.228	-0.566
women	0.601	-0.046	-0.099	0.011	-0.097	-0.605	0.256	-0.551
girls	0.417	-0.005	-0.409	-0.531	-1.319	-0.035	-0.941	-0.361
men	0.778	0.407	0.426	0.534	-0.056	0.518	0.108	0.245
boys	0.897	-0.017	-0.002	-0.182	-1.313	0.449	-0.828	0.521
accessories	0.868	-0.378	-1.248	1.541	0.324	0.283	-0.491	0.081
socks	0.276	0.354	0.186	0.301	-0.643	-0.022	0.321	0.241
luggage	**0.797**	**1.750**	**-2.307**	**-0.560**	**0.031**	**0.921**	**0.417**	**0.313**
dress	**0.282**	**0.233**	**0.043**	**0.175**	**-0.501**	**-0.381**	**0.298**	**-0.026**
baby	0.346	-0.550	-1.136	-0.044	-2.005	0.690	-1.092	0.010
jewelry	-0.316	0.348	-0.309	0.879	-0.766	1.124	-0.080	-2.039
black	0.427	0.030	-0.019	0.224	-0.162	-0.325	0.170	-0.173
boots	1.009	-0.304	0.032	-0.334	-0.096	0.111	0.118	-0.519
shirts	**0.444**	**0.453**	**0.394**	**0.518**	**-0.531**	**0.100**	**0.146**	**0.204**
shirt	**0.329**	**0.422**	**0.227**	**0.456**	**-0.700**	**0.067**	**0.106**	**0.234**
underwear	0.231	0.491	0.226	0.202	-0.774	0.005	0.229	0.310

Figure 10.3: Examples of words converted to numerical features via Word2Vec. Compare embeddings for words "shirt" and "shirts" (highlighted in red) and for "luggage" and "dress" (highlighted in blue). The embeddings for shirt and shirts are much more similar than the embeddings for luggage and dress.

column represents a "trait" and the cell entry represents the loading of the word in the row in that trait. The numbers are more useful in comparison with each other across different rows which allows us to understand word similarity. For example, we can see that the very similar words "shirt" and "shirts" have very similar embeddings while the embeddings for the seemingly relatively different words "luggage" and "dress" are quite dissimilar.

In our context, we can think of each word appearing in a datum (e.g. a product description) as a random variable T and denote its corresponding embedding representation by U.

> One of the ways to train the word embeddings is to predict the middle word from the words that surround it in word sentences.

Given a subsentence s of $K+1$ words, we have a central word $T_{c,s}$ whose identity we would like to predict. As predictors, we have the context words $\{T_{o,s}\}$ that surround the central word $T_{c,s}$. One approach for forming the prediction starts by collapsing the embeddings for context words by a sum,[3]

$$\bar{U}_s = \frac{1}{K} \sum_o U_{o,s},$$

where $U_{o,s}$ is the element of ω corresponding to the word $T_{o,s}$. This step imposes a drastically simplifying assumption that the context words are exchangeable – i.e. the position of each word is not important.

The probability of the middle word $T_{c,s}$ being equal to t is

3: Why not? We can try it and see if it works.

modeled via the multinomial logit function:

$$p_s(t;\pi,\omega) := P\Big(T_{c,s} = t \mid \{T_{o,s}\}, \pi, \omega\Big) = \frac{\exp(\pi_t' \bar{U}_s(\omega))}{\sum_{\bar{t}} \exp(\pi_{\bar{t}}' \bar{U}_s(\omega))},$$

where $\pi = (\pi_1, ..., \pi_d)$ is an $m \times d$ matrix of parameter vectors defining the choice probabilities. The model constrains the choice probabilities π to be ω, and estimates ω using the quasi-maximum likelihood method:

$$\max_{\omega=\pi} \sum_{s \in \mathcal{S}} \log p_s(T_{i,s}; \pi, \omega),$$

where we sum the log-probabilities over many examples $\mathcal{S}$ of subsentences s. Once we are done training, we can generate the embedding for the title or description of product i, containing the embedded words $\{U_{j,i}\}_{j=1}^{J}$ by simply averaging them:

$$W_i = \frac{1}{J} \sum_{j=1}^{J} U_{j,i}. \tag{10.4.1}$$

Remark 10.4.1 In summary, the Word2Vec algorithm transforms text into a vector of numbers that can be used to compactly represent words. The algorithm trains a neural network in a supervised manner such that contextual information is used to predict another part of the text.

For example, let's say that the title description of the item is: "Hiigoo Fashion Women's Multi-pocket Cotton Canvas Handbags Shoulder Bags Totes Purses." The model will be trained using many n-word subsentence examples, such that the center word is predicted from the rest. If we just use $n = 3$ subsentence examples, then we train the model using the following examples: (Hiigoo,Women's) $\rightarrow$ Fashion, (Fashion,Multi-pocket)$\rightarrow$ Women's, (Women's,Cotton) $\rightarrow$ Multi-pocket, and so on.

How do we judge whether the text embedding is successful or not? In the hedonic price context, we can check whether Word2Vec features improve the quality of prediction of the price by the hedonic model. We can also check if similar words T_k and T_l have similar embeddings. We can measure the similarity through cosine similarity:

$$\text{sim}(T_k, T_l) = U_k' U_l / (\|U_k\| \|U_l\|) \in [-1, 1].$$

The more similar the words are, according to our human notion of similarity, the higher the value our formal measure of similarity should take. For example, the following are the two words that are most similar to "tie" under the similarity measure: "necktie" and "bowtie." The embeddings also induces an interesting vector space on the set of words, which seems to encode analogues well. For example, the word "briefcase" is very cosine-similar to the artificial latent word[4]

$$\text{Artificial word} = \text{Word2Vec(men's)} + \text{Word2Vec(handbag)} - \text{Word2Vec(women's)}.$$

4: This example also shows us how word embeddings very easily encode and propagate biases that exist in document corpora that are typically used in machine learning; a realization that has been highlighted by several recent works [9]. One should always be cognizant of such inherent biases in trained embeddings. Recent works in machine learning (e.g. [9]) provide automated approaches that partially correct for these biases, though not completely removing the problem [10].

This similarity between a real word and our constructed latent word gives some justification for the "averaging" of embeddings to summarize whole sentences or descriptions.

Word2vec embeddings were among the first generation of early successful embedding algorithms. These algorithms have been improved by the next generation of NLP algorithms, such as ELMo and BERT, which are discussed next.

Second Generation: Sequence Models

A major advance in language modeling has been to represent text as a sequence using recurrent (autoregressive) models. Among various benefits, representing text with an autogressive structure allows for better capturing the context around words.

Of note is the Embeddings from Language Models (ELMo) algorithm [11], which uses the idea of the Shannon game where we aim to guess a word in a sentence, m, consisting of n total words. Specifically, we consider the problem of predicting word $k+1$ using the preceding k words via

$$p^f_{k,m}(t) = P(T_{k+1,m} = t \mid T_{1,m}, ..., T_{k,m}; \theta)$$

and similarly consider the reverse prediction via

$$p^b_{k,m}(t) = P(T_{k-1,m} = t \mid T_{k,m}, ...T_{n,m}; \theta),$$

where θ is a parameter vector. ELMo then uses recurrent neural networks (RNNs) to model these probabilities.

On Recurrent Neural Networks A RNN is a particular architecture for sequence input and output where we use

neurons from the previous prediction to make the current prediction.RNNs are essentially the neural network version of linear autoregressive models, such as ARIMA models, which go back to the early work of statisticians George Box and Gwilym Jenkins [12, 13] and have also been used in economics to model volatility of financial assets in the GARCH model of economist Tim Bollerslev [14].

In its simplest form, a RNN parses inputs in a serial manner $T_1, \ldots, T_t, \ldots, T_k$ where each step t produces a state vector $S_t = \sigma(AT_t + BS_{t-1} + c)$ that is a non-linear function (a set of neurons) of the current input and the previous state vector. That is, σ is an activation function as presented in Section 8.3 applied elementwise to each coordinate. Moreover, a RNN produces an output prediction vector $y_t = \sigma(DS_t + e)$ that is a non-linear function (a set of neurons) of the current state. The parameters A, B, c, D, e of all these neurons are the same (shared) across steps.

Parameters are estimated by maximizing parameterized approximate versions of the log-likelihoods of the observed data (aka quasi-likelihoods), typically referred to as quasi-maximum log-likelihood methods, where the forward and backward log quasi-likelihoods are added together.

Specifically, ELMo uses a particular form of recursive neural network called Long Short-Term Memory (LSTM) network. LSTMs improve upon the numerical stability of RNNs by allowing for the "state" to pass through the current step as-is, without any non-linearity applied. Allowing the state to pass through steps without alteration helps in propagating information across distant steps and thus better accommodates long-term memory.

To give a simple example, suppose we wanted model word choice with a multinomial logit function, as in the previous subsection, but wanted to better grasp the positional context of the individual words. Rather than start by collapsing the embeddings for context words surrounding a target central word via a sum, we could instead keep track of word order and assign individual parameters to each context. For example, we could model the forward predicted probability of word k in sentence m as

$$P(T_{k,m} = t \mid \{T_{j,m}\}_{j=1}^{k-1}, \pi) = \frac{e^{\sum_{j=1}^{k-1} \pi'_{t,j} U_{j,m}(\omega)}}{\sum_{\bar{t}} e^{\sum_{j=1}^{k-1} \pi'_{\bar{t},j} U_{j,m}(\omega)}},$$

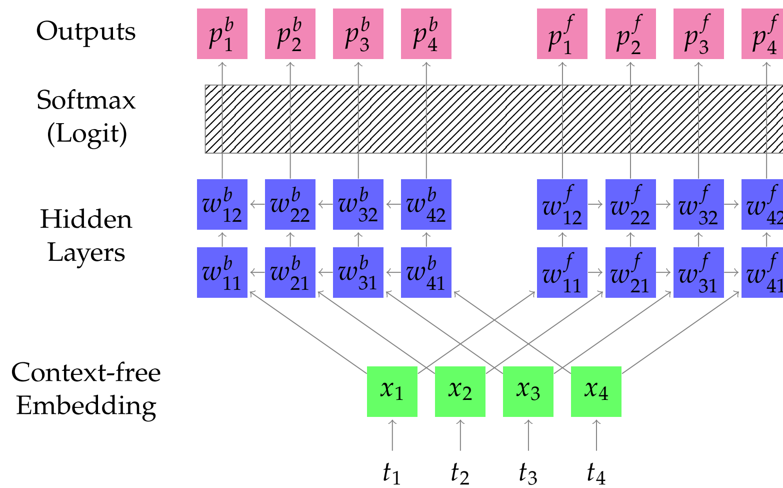

Figure 10.4: ELMo Architecture. ELMo network for a string of 4 words, with $L = 2$ hidden layers. Here, the softmax layer (multinomial logit) is a single function mapping each input in $\mathbb{R}^d$ to a probability distribution.

and similarly model the reverse prediction problem where we recall that $U_{j,m}(\omega)$ is the embedding corresponding to word j in sentence m. ELMo uses a more sophisticated (and more parsimonious) recursive nonlinear regression model (specifically a recurrent neural network) to build these probabilities. We illustrate a simple ELMo structure in Figure 10.4.

The basic structure of ELMo.
Given a sentence m of n words,

1. Words are mapped to context-free embeddings in $\mathbb{R}^d$.

2. A network is trained to predict each word $T_{k,m}$ of a string given (a) words $(T_{1,m}, \ldots, T_{k-1,m})$ (forward prediction) and (b) words $(T_{k+1,m}, \ldots, T_{n,m})$ (backward prediction). The objective is to maximize the average over the sum of the log-likelihoods of the $2n - 2$ words being predicted, where the average is taken over all sentences.

3. The embedding of word $T_{k,m}$ is given by a weighted average of outputs of certain hidden neurons corresponding to this word's entire context. Importantly, a subset of the parameters is coupled across the forward and backward prediction problems (2a) and (2b). In particular, the first layer that goes out of the context-free embedding and the final ("softmax") layer that produces the probabilistic predictions is the same for the two prediction objectives (2a) and (2b). Thus the inputs to this layer, which represent the forward and

A softmax layer assigns probabilities to each class in a multi-class problem. It is a multi-class generalization of logistic regression that assumes mutually exclusive classes.

backward context, are constrained to lie in "the same space."

Training

In Figure 10.4, the output probability distribution p_k^f is taken as a prediction of $T_{k+1,m}$ using words $(T_{1,m}, \dots, T_{k,m})$. Similarly, p_k^b is taken as a prediction of $T_{k-1,m}$ using words $(T_{k,m}, \dots, T_{n,m})$. The parameters of the network, θ, are obtained by maximizing the quasi-log-likelihood:

$$\max_{\theta} \sum_{m \in \mathcal{M}} \left(\sum_{k=1}^{n-1} \log p_{k,m}^f (T_{k+1,m}; \theta) + \sum_{k=2}^{n} \log p_{k,m}^b (T_{k-1,m}; \theta) \right),$$

where $\mathcal{M}$ is a collection of sentences. In our running pricing example, $\mathcal{M}$ would be the collection of titles and product descriptions taken from product web pages.

Producing embeddings

To produce embeddings from the trained network, each word t_k in a sentence $m = (t_1, ..., t_n)$ is mapped to a weighted average of the outputs of the hidden neurons indexed by k:

$$t_k \mapsto w_k := \sum_{i=1}^{L} (\gamma_i w_{ki}^f + \bar{\gamma}_i w_{ki}^b).$$

The embedding for the sentence (or an entire product description in our example) is produced by summing the embeddings for each individual word. The weights γ and $\bar{\gamma}$ can be tuned by the neural network performing the final task. In principle, the whole network could be plugged in to the network performing the final task and allowed to update. However, the ELMo architecture and methodology is more inline with being used as a feature extractor, with only the final linear layer being trained towards the target task (in sharp contrast to the BERT model that we outline next; see e.g. [15]).

Third generation: Transformers

A subsequent major advance in language modeling has been the development and use of the transformer architecture. Going beyond backwards and forwards sequences, transformers use a

mechanism termed "self-attention" [16] in order to model the importance of different parts of the text in understanding any one other part. Like RNNs, this self-attention mechanism allows for understanding context; but unlike RNNs, it allows the model to better focus on potentially far away parts of the text that may be more relevant than nearby words for understanding context. For example, looking beyond the local neighborhood of say the word "it" allows understanding that "it" in one sentence refers to a particular word from a previous sentence.

An early and prominent example of transformer-based language models is Bidirectional Encoder Representations from Transformers (BERT) [17].

Unlike the language model in ELMo which predicts the next word from previous and subsequent words, the BERT model is trained on two self-supervised tasks simultaneously:

- Mask Language Model: Randomly mask a certain percentage of the words in a sentence and predict the masked words.
- Next Sentence Prediction: Given a pair of sentences, predict whether one sentence precedes another.

The basic structure of BERT.

1. Each word in the input sentence is broken into subwords (tokenized) and each piece is called a "token." Each token is encoded using a context-free embedding called WordPiece. A special token [cls] is added to the beginning of the sequence. x% of the tokens representing individual words are replaced by [mask].

2. For each token, its input representation consists of i) its token embedding from (1), ii) its position embedding indicating the position of the token in the sentence, and iii) its segment embedding indicating whether it belongs to sentence A or B.

3. The input representation of tokens in the sequence is fed into the main model architecture: L layers of Transformer-Encoder blocks. Each block consists of a "multi-head attention layer" (described below), followed by a feed forward layer.

4. The output representation of the mask token [mask] is used to predict the masked word via a softmax layer, and the output representation of the special [cls] token is used for Next Sentence Prediction. The loss function is a combination of the two losses.

We next focus in detail on the main structure used to construct the network in (3), especially the "multi-head attention" layer.

Computing the Attention

We begin with the context-free embeddings $(x_1, x_2, \ldots, x_n)$, for n words, with each $x_k \in \mathbb{R}^d$. Let X denote the matrix whose k-th row is the embedding x_k. An attention module transforms this matrix of n embeddings, X, into another matrix of n embeddings, where each row k of the new matrix contains an embedding of the "information" in a "neighborhood" around token k. The notion of "neighborhood" and the notion of "information" are all parameterized by neural network parameters of the attention module and learnable in a data-driven manner as we describe below.

The goal of an attention module is to create weighted neighborhoods (*attention regions*) of seemingly distant tokens in a data-driven manner and then create embeddings that correspond to linear combinations of the embeddings of the tokens in these attention regions. One way to achieve this goal is to decouple the "neighborhood" representation of a token with the representation of each "meaning" or "value." Thus we will transform each token embedding x_k into a *key embedding* $\kappa_k := x_k'\omega^K$, where $\omega^K \in \mathbb{R}^{d \times d_k}$ is a learnable matrix parameter, and a *value embedding* $v_k := x_k'\omega^V$, where $\omega^V \in \mathbb{R}^{d \times d_v}$ is a learnable matrix parameter. Then a neighborhood can be encoded by a *query* vector q that lies in the same space as the space of keys and such that the weighted neighborhood is defined via a similarity metric between the vector q and the key vectors. Attention mechanisms used in Transformers use a scaled inner product as the similarity, i.e. $s_k := q'\kappa_k/\sqrt{d_k}$. Then this similarity is passed through a soft-max function $\sigma(\cdot)$ to map it to a selection probability in $[0, 1]$. Finally, as we alluded to in the beginning the *embedding of the neighborhood* that corresponds to this query q is simply the weighted average of the value embeddings of the tokens, i.e. $a := \sum_{k=1}^{n} \sigma(s_k)v_k$.

Suppose now that we had n neighborhood queries $q_1, \ldots, q_n$, then we could create n such neighborhood embeddings $a_1, \ldots, a_n$. Transformers consider "self-attention" queries, where each of the n queries q_k corresponds to a *query embedding* associated with a particular token and is yet another linear embedding of the form $q_k = x_k' \omega^Q$, where $\omega^Q \in \mathbb{R}^{d \times d_k}$ is a learnable matrix parameter. Then for each such query we can calculate the corresponding *neighborhood embedding* a_k.

Overall this transformation takes as input a matrix $X \in \mathbb{R}^{n \times d}$, where each row corresponds to an original token embedding and transforms it into a matrix A, where each row k corresponds to the neighborhood embedding associated with query q_k, which in turn is associated with token x_k. We can write this calculation in matrix form: Let $Q = X\omega^Q$ denote the matrix with rows corresponding to query embeddings, let $K = X\omega^K$ denote the matrix with rows corresponding to key embeddings, and let $V = X\omega^V$ denote the matrix with rows corresponding to value embeddings. Then the attention embeddings (or neighborhood embeddings) can be written in matrix form as

$$A = \text{Attention}(Q, K, V) := \sigma\left(QK^\top / \sqrt{d_k}\right) V.$$

A Multi-Head Attention mapping, which is the building block of the BERT model, builds many such attention transformations, for different matrix parameters $\{\omega_j^Q, \omega_j^K, \omega_j^V\}_{j=1}^h$, calculates the corresponding attention embedding matrices $A_j \in \mathbb{R}^{n \times d_v}$, then concatenates the results in a big embedding matrix $A = \text{Concatenate}(A_1, \ldots, A_h) \in \mathbb{R}^{n \times h \cdot d_v}$ and applies a linear projection transformation $A\omega^O$, where $\omega^O \in \mathbb{R}^{h \cdot d_v \times d_o}$, to produce the final output encoding. Thus, we can define the basic Multi-Head Attention transformation:

$$X \longmapsto \text{MultiHead}(X) := \text{Concatenate}(\text{Head}_1, \ldots, \text{Head}_h)\omega^O,$$

$$\text{Head}_j = \text{Attention}(X\omega_j^Q, X\omega_j^K, X\omega_j^V),$$

Each Transformer building block in BERT consists of a series of several repetitions of multi-head attention encodings, followed by a fully connected neural network (applied to each of the n output encodings separately). The input n encodings of each repetition is the output of the previous repetition.

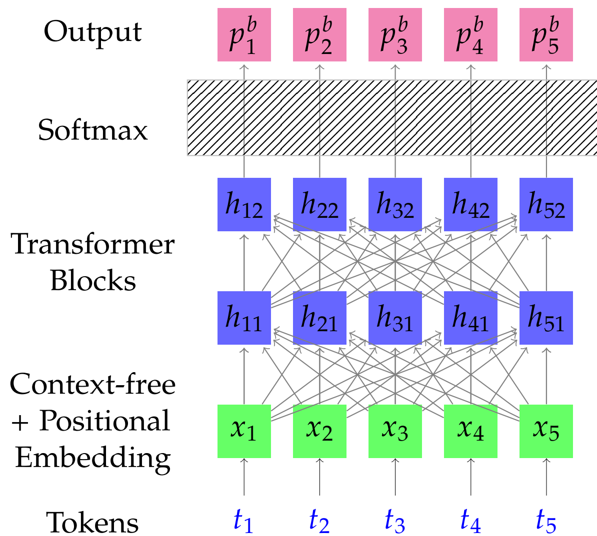

Figure 10.5: BERT Architecture

Generating product embeddings

Depending on specific tasks and resources, Devlin et al. [17] suggested to construct BERT embeddings in various ways:

- Use the last layer, second-to-last layer, or concatenate the last 4 layers of the encoder outputs from the pre-trained BERT model.
- Fine tune the whole BERT model using the downstream task.
- Train the BERT language model from scratch on new data.

Tuning only a final linear layer on top of a pre-trained embedding network and freezing all other parameters of the embedding is referred to in the machine learning literature as *linear probing*. If one allows for the parameters of the embedding itself to be updated when optimizing for a particular downstream prediction task, then this practice is referred to as *fine-tuning*. [18] presents a way of blending the two modes by first training the final linear layer and then un-freezing the remaining parameters of the embedding and continuing to train. This blending seems to produce substantial gains in generalization ability and accuracy of the resulting predictive model.

In the hedonic price example discussed in Section 10.6, the feature-based approach was chosen. Specifically, the second-to-last layer from a pre-trained BERT model was extracted as embeddings to be used as covariates in the final price prediction task. Each product's text embedding is the average of the embeddings of each word/token from the input text field.

Beyond ELMo and BERT

ELMo and BERT are both important breakthroughs in NLP. The former marked the first contextual word embedding trained from a deep language model, and the latter was the first contextual word embedding using Transformer architecture. The biggest difference lies in the choice of fundamental architectures: ELMo is based on a Recurrent Neural Network (RNN), while BERT is based on the Transformer architecture. RNNs can struggle to capture long-term dependencies, whereas the

Transformer architecture is more efficient at capturing long-range dependencies in the text. Furthermore, ELMo creates context by using the left-to-right and right-to-left language model representations, while BERT models the entire context simultaneously.

Large language models are continuously evolving and becoming ever more powerful and sophisticated in their understanding of language and meaning. The latest generation of large language models lie in the Generative Pre-trained Transformer (GPT) family [19–21]. While BERT can be understood to use the transformer architecture as an *encoder*, GPT models use the transformer architecture in a *decoder* for a generative model of the probabilities in an autoregressive model reminiscent of the one used in ELMo. GPT models combine these modeling ideas from their successful precursors with pre-training on a large corpus of text. With the rapid development in the space of large language and multi-modal models, the latest and greatest models will certainly advance beyond the descriptions in this book, but the principles of using these models to understand complex data and use it to support robust causal inference will likely remain the same.

Revisiting the Price Elasticity for Toy Cars

In Chapter 0, Chapter 4, and Chapter 9, we saw how using increasingly flexible learning methods (OLS, LASSO, nonlinear regression) to control for confounding in the price-sales data for toy cars lead to increasingly more negative estimates and confidence intervals for elasticity. However, the models discussed in those chapters only used the categorical (brand, subcategory) and numeric (physical dimensions) features of the products. However, we actually observe much richer data: all the text on the product page, including the product description. Text embeddings are a great way to leverage these data and include them in a causal analysis of price elasticity. We can take BERT and plug it into neural networks, one to predict price and one for sales: we take the text as input, pass it through BERT initialized at the pre-trained model, add an additional dense layer, and train the whole network. Doing this over 5 folds, we obtain a cross-validated R^2 of 0.55 for predicting Y and 0.027 for predicting D, improving upon the best nonlinear methods considered in the Chapter 9. Applying DML with these new predictors leads to a point estimate for elasticity of -0.174 and 95% confidence interval of [-0.214, -0.135]. The further increase in the magnitude of the estimated coefficient suggests we are

better controlling for observed confounders by including the text data as the more negative elasticity estimates align more closely with our theoretical prediction.

At the end of the chapter we provide a notebook wherein we repeat the exercise of constructing neural nets using BERT for predicting Y and D and plug them into DML. The results in that notebook are different than those reported here (and previously) as we use a publicly available dataset in that notebook as opposed to the proprietary data underlying the numbers we report here.

The biggest difference between the public and private data is that the public data does not have the same range of numeric features as in the private data.

10.5 Image Embeddings

One of the most successful deep learning models for image classification was the ResNet50 model developed by He et al. [22]. At the time of the release, the paper achieved the best results in image classification, in particular for the ImageNet and COCO datasets.

The central idea of the paper is to exploit "partial linearity": traditional nonlinearly-generated neurons are combined (or added together) with the previous layer of neurons. More specifically, ResNet50 takes a standard feed-forward convolutional neural network and adds skip connections that bypass two (or one or several) convolutional layers at a time. Each skipping step generates a residual block in which the convolution layers predict a residual.

Formally, each k-th residual block is a neural network mapping

$$v \longmapsto (v, \sigma_k^0(\omega_k^0 v)) \longmapsto (v, \sigma_k^1 \circ \omega_k^1 \sigma_k^0(\omega_k v)) \\ \longmapsto v + \sigma_k^1 \circ \omega_k^1 \sigma_k^0(\omega_k^0 v),$$

where ω's are matrix-valued parameters or "weights." This structure can be seen as a special case of general neural network architecture, designed so that it is easy to learn the identity sub-maps (entering the composition of the entire network). Putting together many blocks like these sequentially results in the overall architecture depicted in Figure 10.6.

The deep feed-forward convolutional networks developed in prior work suffered from major optimization problems – once the depth was sufficiently high, additional layers often resulted in much higher validation and training error. It was argued that this phenomenon was a result of "vanishing gradients," where

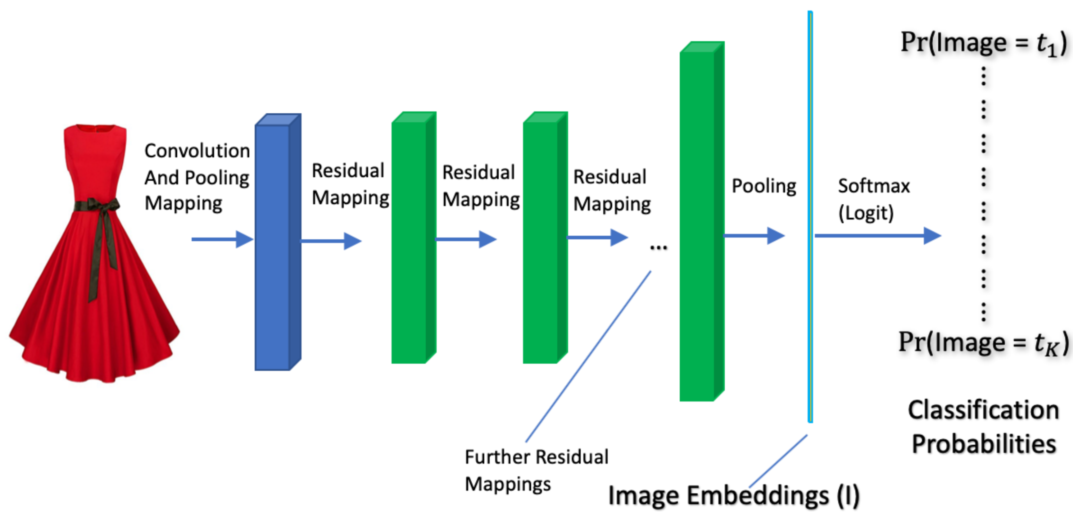

Figure 10.6: The ResNet50 operates on numerical 3-dimensional arrays representing images. It first does some pre-processing by applying convolutional and pooling filters, then it applies many L-residual block mappings, producing the arrays shown in green. The penultimate layer produces a high-dimensional vector I, the image embedding, which is then used to predict the image type.

in a network of n layers, computation by backpropogation using the chain rule involves multiplying n small numbers (if using traditional activation functions, recent popular activation functions such as RELU do not induce such a small derivative), causing the gradient to "vanish" for early layers and posing a computational challenge. The residual network architecture addresses this by using the residual block architecture: including the residual directly via skip connections reduces the minimizing impact of the activation function. The creation of this architecture has allowed for high quality training even for very deep networks.

In many applications, we will not be interested in the final predictions from the image classification task. Rather, we will be interested in using lower levels of the network, such as the last hidden layer, as our image embeddings to be used as inputs into the modeling task of interest. For example, in the pricing example discussed next in Section 10.6, we feed image data from product webpages into a publicly trained ResNet50 model and extract the final layer to generate the image embeddings.

10.6 Application: Hedonic Prices

Here we apply our new knowledge of embeddings to review an empirical application considered in Bajari et al. [23]. The

application is a prediction problem which deals with hedonic price models. An empirical hedonic model is a predictive model for price given a traded object's characteristics.[5] Here, the goal is to predict the price of apparel bought and sold on Amazon.com using the product's image and description:

5: It can be given structural or causal interpretation using the so-called hedonic price models from economics.

$$P_{it} = H_{it} + \epsilon_{it} = h_t(X_{it}) + \epsilon_{it}, \quad \mathrm{E}[\epsilon_{it} \mid X_{it}] = 0, \tag{10.6.1}$$

where P_{it} is the price of product i at time t (in months), X_{it} are the product features, and the price function $x \mapsto h_t(x)$ can change from period to period, reflecting the fact that product attributes/features may be valued differently in different periods. [23] use the data from time period t to estimate the function h_t using deep neural network methods. The results are contrasted with classical linear regression methods as well as other modern regression methods, such as the random forest.

One of the main uses of hedonic prices is construction of cost of living indices. The use of hedonic prices allows us to "price" the product attributes as well as entire "baskets of attributes" that consumers buy. Then, given a reference "basket of attributes," one can look at the hedonic cost of a basket today compared to its cost in an earlier reference period to determine whether the cost increased or decreased. These types of calculations underlie the construction of commonly used consumer price indices (measuring inflation rates), at least for categories such as apparel products.

A key component of the approach taken in [23] is the use of product features X_{it} generated as neural network embeddings of text and image information about the product. Specifically, X_{it} consists of text embedding features W_{it}, constructed by converting the title and product description available on a product's web page into numeric vectors, and image embedding features I_{it} constructed by converting the product image into numeric vectors:

$$X_{it} = (W_{it}', I_{it}')'. \tag{10.6.2}$$

These text and image embedding features are generated respectively by applying the BERT and ResNet50 mappings.

The model takes high-dimensional text and image features as inputs, converts them into a lower dimensional vector of value embeddings using deep learning methods, and outputs simultaneous predictions of price in all time periods.

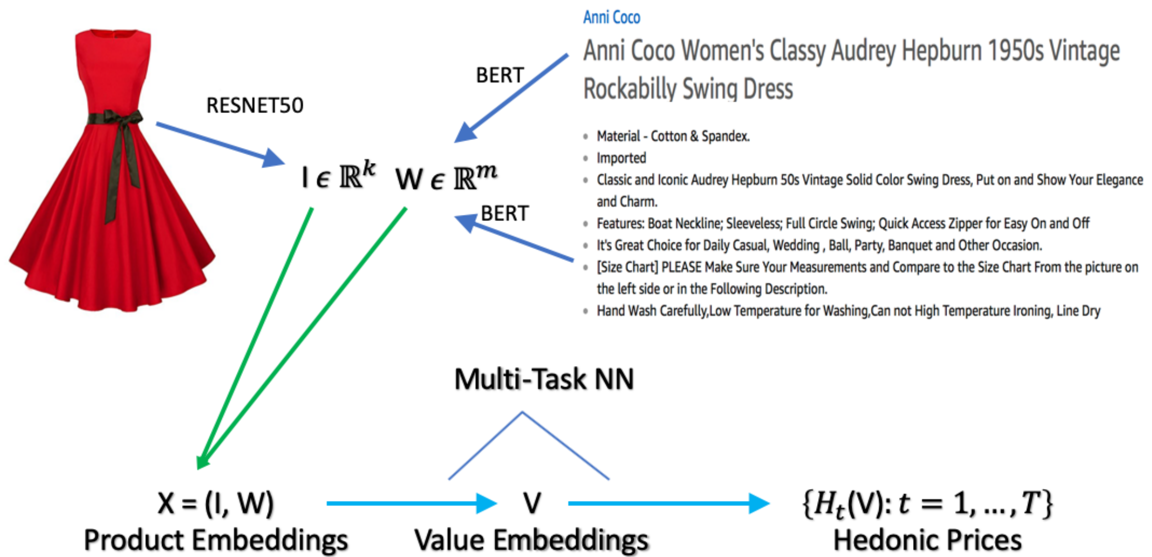

Figure 10.7: The structure of the predictive model in Bajari et al. [23]. The input consists of images and unstructured text data. The first step of the process creates numerical embeddings I and W for images and text data via deep learning methods, specifically ResNet50 and BERT. The second step of the process takes as its input $X = (I, W)$ and creates predictions for hedonic prices $H_t(X)$ using deep learning methods with a multi-task structure. The models of the first step are trained on tasks unrelated to predicting prices (e.g., image classification or word prediction), where embeddings are extracted as hidden layers of the neural networks. The models of the second step are trained by price prediction tasks. The multitask price prediction network creates an intermediate lower dimensional embedding $V = V(X)$, called a value embedding, and then predicts the final prices in all time periods $\{H_t(V), t = 1, ..., T\}$. Some variations of the method include fine-tuning the embeddings produced by the first step to perform well for price prediction tasks (i.e. optimizing the embedding parameters so as to minimize price prediction loss).

The general structure of the model takes the form

$$
\begin{aligned}
Z_{it} = \begin{bmatrix} \text{Text}_{it} \\ \text{Image}_{it} \end{bmatrix} &\overset{e}{\longmapsto} X_{it} \\
&\overset{g_1}{\longmapsto} E^{(1)}_{it} \dots \overset{g_m}{\longmapsto} E^{(m)}_{it} =: V_{it} \overset{\theta'}{\longmapsto} \{H_{it}\}_{t=1}^{T} := \{\beta_t' V_{it}\}_{t=1}^{T}.
\end{aligned}
$$

Here Z_{it},[6] the original input which lies in a very high-dimensional space, is nonlinearly mapped into an embedding vector X_{it} which is of moderately high dimension (up to 5120 dimensions in this example). X_{it} is then further nonlinearly mapped into a lower dimension vector $E^{(1)}_{it}$. This process is repeated to produce the final hidden layer, $V_{it} = E^{(m)}_{it}$, which is then linearly mapped to the final output that consists of hedonic price H_{it} for product i in all time periods $t = 1, ..., T$.

6: As a practical matter, most of the product attributes in [23] are time-invariant - that is, $Z_{it} = Z_i$ has no time variation. We state the model in more generality here.

The last hidden layer $V = E^{(m)}$ is called the *value embedding* in this context – the value embedding represents latent attributes to which dollar values are attached. The embeddings produced in this example are moderately high-dimensional (up to 512

dimensions) summaries of the product, derived from the most common attributes that directly determine the price of the predicted hedonic price of the product. Note that the embeddings V in this example do not depend on time and so may be thought of as representing intrinsic, potentially valuable attributes of the product. However, the predicted price does depend on time t via the coefficient β_t, reflecting the fact that the different intrinsic attributes are valued differently across time.

The network mapping above comprises a deep neural network with neurons $E_{k,\ell}$ of the form

$$g_\ell : v \longmapsto \{E_{k,\ell}(v)\}_{k=1}^{K_\ell} := \{\sigma_{k,\ell}(v'\alpha_{k,\ell})\}_{k=1}^{K_\ell}. \tag{10.6.3}$$

Here $\sigma_{k,\ell}$ is the activation function that can vary with the layer ℓ and can vary with k, from one neuron to another.

The model is trained by minimizing the loss function

$$\min_{\eta \in \mathcal{N}, \{\beta_t\}_{t=1}^T} \sum_t \sum_i (P_{it}^c - \beta_t' V_{it}(\eta))^2 Q_{it}, \tag{10.6.4}$$

where η denotes all of the parameters of the mapping

$$X_{it} \mapsto V_{it}(\eta)$$

and $\mathcal{N}$ represents the parameter space. Here, we are using a weighted loss where we weight by the quantity of product i sold at time t, Q_{it}.

Next we review how the initial embedding is generated. A multilingual BERT model is used to convert text information and the ResNet50 model is used to convert images into a subvector of $E_{it}^{(1)}$. These models are trained on auxiliary prediction tasks with auxiliary outputs $A_{T_{it}}$ for text and $A_{I_{it}}$ for images. Introducing these auxiliary tasks can be illustrated diagrammatically as

$$X_{it} = \begin{bmatrix} \text{Text}_{it} \\ \text{Image}_{it} \end{bmatrix} \overset{e}{\longmapsto} \begin{matrix} A_{T_{it}} \\ \uparrow \\ W_i \\ I_i \\ \downarrow \\ A_{I_{it}} \end{matrix} =: E_{it}^{(1)} \ldots \longmapsto E_{it}^{(m)} := V_{it} \longmapsto \{\hat{P}_{it}^*\}_{t=1}^T, \tag{10.6.5}$$

The embeddings W_{it} and X_{it} forming $E_{it}^{(1)}$ are obtained by mapping them into auxiliary outputs A_{Tj} and A_{Ij} that are scored on natural language processing tasks and image classification tasks respectively. This step uses data that are not related to prices. The parameters of the mapping generating $E_{it}^{(1)}$ are considered as fixed in our analysis.

The price prediction network we employ in this example contains three hidden layers, with the last hidden layer containing 400 neurons. The network is trained on a large data set with more than 10 million observations. A large enough data set is crucial for training successful neural networks.

The accuracy of prediction as measured by the R^2 on the test sample is about 90%. In contrast, random forests using embeddings deliver an R^2 in the ballpark of 80%; the linear model using least squares applied to embeddings delivers an R^2 in the ballpark of 70% and the linear model using only simple catalog features (without embeddings) delivers an R^2 lower than 40%.

Thus, embeddings offer a means of making use of complex data for predictions and, at least for large data sets, neural nets can offer predictive improvements relative to competing machine learning approaches.

10.7 Notes

[24] develop "DoubleMLDeep" which explicitly explores deep neural network architectures for incorporating text and image data as confounding variables in the DML framework.

10.8 Notebooks

Notebook 10.8.1 (Autoencoders) Python Autoencoders Notebook and R Autoencoders Notebook provide an introduction to autoencoders, starting from classical principal components.

Notebook 10.8.2 (Embeddings via BERT) Python Toys and Prices Notebook provides an introduction to text embeddings via BERT and provides an application to predicting demand for toys.

10.9 Exercises

Exercise 10.9.1 (Autoencoders) Work through the Autoencoders notebook. Try to improve the performance of the autoencoders. Report your findings (even if you don't manage to improve them! :-)).

Exercise 10.9.2 (BERT) Work through the BERT notebook. Try to experiment with the structure of the neural nets and demand estimation procedure. Report your findings.

Bibliography

[1] Susan Sontag. 'Turning Points'. In: *Irish Pages* 2.1 (2003), pp. 186–191 (cited on page 268).

[2] Matthew Gentzkow, Bryan Kelly, and Matt Taddy. 'Text as Data'. In: *Journal of Economic Literature* 57.3 (2019), pp. 535–74. DOI: 10.1257/jel.20181020 (cited on page 270).

[3] Pierre Comon. 'Independent component analysis, A new concept?' In: *Signal Processing* 36.3 (1994). Higher Order Statistics, pp. 287–314. DOI: https://doi.org/10.1016/0165-1684(94)90029-9 (cited on page 274).

[4] Francesco Locatello, Stefan Bauer, Mario Lucic, Gunnar Raetsch, Sylvain Gelly, Bernhard Schölkopf, and Olivier Bachem. 'Challenging Common Assumptions in the Unsupervised Learning of Disentangled Representations'. In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 4114–4124 (cited on page 274).

[5] Diederik P. Kingma and Max Welling. 'Auto-Encoding Variational Bayes'. In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2014 (cited on page 274).

[6] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 'Representation learning: A review and new perspectives'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.8 (2013), pp. 1798–1828 (cited on page 275).

[7] Diederik P Kingma, Max Welling, et al. 'An introduction to variational autoencoders'. In: *Foundations and Trends® in Machine Learning* 12.4 (2019), pp. 307–392 (cited on page 275).

[8] Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 'Efficient Estimation of Word Representations in Vector Space'. In: *International Conference on Learning Representations*. 2013 (cited on page 276).

[9] Tolga Bolukbasi, Kai-Wei Chang, James Zou, Venkatesh Saligrama, and Adam Kalai. 'Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings'. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS'16. Barcelona, Spain: Curran Associates Inc., 2016, 4356–4364 (cited on page 279).

[10] Hila Gonen and Yoav Goldberg. 'Lipstick on a pig: Debiasing methods cover up systematic gender biases in word embeddings but do not remove them'. In: *arXiv preprint arXiv:1903.03862* (2019) (cited on page 279).

[11] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 'Deep contextualized word representations'. In: *CoRR* abs/1802.05365 (2018) (cited on page 279).

[12] George EP Box and Gwilym M Jenkins. *Time Series Analysis Forecasting and Control*. Tech. rep. WISCONSIN UNIV MADISON DEPT OF STATISTICS, 1970 (cited on page 280).

[13] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015 (cited on page 280).

[14] Tim Bollerslev. 'A conditionally heteroskedastic time series model for speculative prices and rates of return'. In: *Review of Economics and Statistics* (1987), pp. 542–547 (cited on page 280).

[15] Matthew E. Peters, Sebastian Ruder, and Noah A. Smith. 'To Tune or Not to Tune? Adapting Pretrained Representations to Diverse Tasks'. In: *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*. Florence, Italy: Association for Computational Linguistics, Aug. 2019, pp. 7–14. DOI: 10.18653/v1/W19-4302 (cited on page 282).

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 'Attention is All you Need'. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017 (cited on page 283).

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding'. In: *CoRR* abs/1810.04805 (2018) (cited on pages 283, 286).

[18] Ananya Kumar, Aditi Raghunathan, Robbie Jones, Tengyu Ma, and Percy Liang. 'Fine-Tuning can Distort Pretrained Features and Underperform Out-of-Distribution'. In: *International Conference on Learning Representations*. 2022 (cited on page 286).

[19] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 'Improving language understanding by generative pre-training'. In: (2018) (cited on page 287).

[20] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 'Language models are unsupervised multitask learners'. In: *OpenAI blog* 1.8 (2019), p. 9 (cited on page 287).

[21] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 'Language models are few-shot learners'. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1877–1901 (cited on page 287).

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 'Deep residual learning for image recognition'. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (cited on page 288).

[23] Patrick L. Bajari, Zhihao Cen, Victor Chernozhukov, Manoj Manukonda, Jin Wang, Ramon Huerta, Junbo Li, Ling Leng, George Monokroussos, Suhas Vijaykunar, et al. *Hedonic prices and quality adjusted price indices powered by AI*. Tech. rep. cemmap working paper CWP04/21, 2021 (cited on pages 289–291).

[24] Sven Klaassen, Jan Teichert-Kluge, Philipp Bach, Victor Chernozhukov, Martin Spindler, and Suhas Vijaykumar. *DoubleMLDeep: Estimation of Causal Effects with Multimodal Data*. 2024 (cited on page 293).

Topics