The Hundred-Page Machine Learning Book

Andriy Burkov

"All models are wrong, but some are useful."
— George Box

The book is distributed on the "read first, buy later" principle.

7 Problems and Solutions

7.1 Kernel Regression

We talked about linear regression, but what if our data doesn't have the form of a straight line? Polynomial regression could help. Let's say we have a one-dimensional data $\{(x_i, y_i)\}_{i=1}^N$. We could try to fit a quadratic line $y = w_1 x_i + w_2 x_i^2 + b$ to our data. By defining the mean squared error cost function, we could apply gradient descent and find the values of parameters w_1, w_2, and b that minimize this cost function. In one- or two-dimensional space, we can easily see whether the function fits the data. However, if our input is a D-dimensional feature vector, with $D > 3$, finding the right polynomial would be hard.

Kernel regression is a non-parametric method. That means that there are no parameters to learn. The model is based on the data itself (like in kNN). In its simplest form, in kernel regression we look for a model like this:

$$f(x) = \frac{1}{N}\sum_{i=1}^{N} w_i y_i, \text{ where } w_i = \frac{N k(\frac{x_i - x}{b})}{\sum_{k=1}^{N} k(\frac{x_k - x}{b})}. \tag{1}$$

The function $k(\cdot)$ is a kernel. It can have different forms, the most frequently used one is the Gaussian kernel:

$$k(z) = \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-z^2}{2}\right).$$

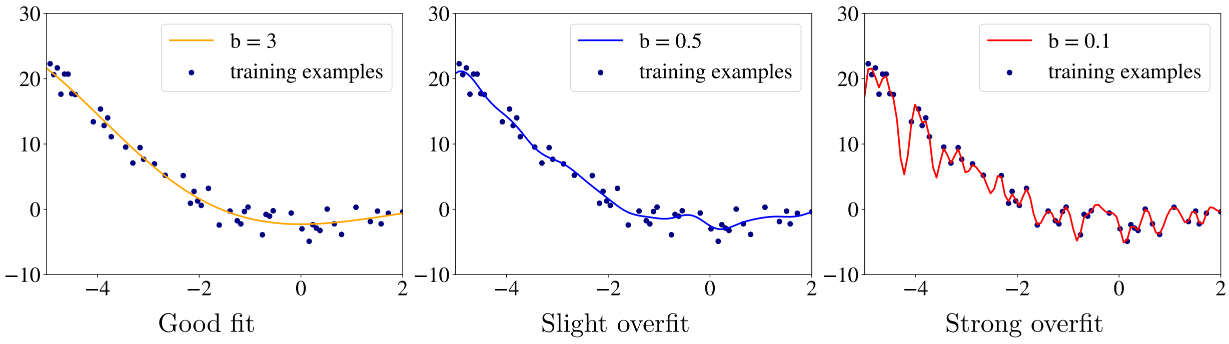

Figure 1: Example of kernel regression line with a Gaussian kernel for three values of b.

The value b is a hyperparameter that we tune using the validation set (by running the model built with a specific value of b on the validation set examples and calculating the mean squared error). You can see an illustration of the influence b has on the shape of the regression line in fig. 1.

If your inputs are multi-dimensional feature vectors, the terms $x_i - x$ and $x_k - x$ in eq. 1 have to be replaced by Euclidean distance $\|\mathbf{x}_i - \mathbf{x}\|$ and $\|\mathbf{x}_k - \mathbf{x}\|$ respectively.

7.2 Multiclass Classification

In multiclass classification, the label can be one of the C classes: $y \in \{1, \ldots, C\}$. Many machine learning algorithms are binary; SVM is an example. Some algorithms can naturally be extended to handle multiclass problems. ID3 and other decision tree learning algorithms can be simply changed like this:

$$f_{ID3}^S \stackrel{\text{def}}{=} \Pr(y_i = c|\mathbf{x}) = \frac{1}{|S|} \sum_{\{y \mid (\mathbf{x}, y) \in S, y = c\}} y,$$

for all $c \in \{1, \ldots, C\}$.

Logistic regression can be naturally extended to multiclass learning problems by replacing the sigmoid function with the **softmax function** which we already saw in Chapter 6.

The kNN algorithm is also straightforward to extend to the multiclass case: when we find the k closest examples for the input $\mathbf{x}$ and examine them, we return the class that we saw the most among the k examples.

SVM cannot be naturally extended to multiclass problems. Some algorithms can be implemented more efficiently in the binary case. What should you do if you have a multiclass problem but a binary classification learning algorithm? One common strategy is called **one versus rest**. The idea is to transform a multiclass problem into C binary classification problems and build C binary classifiers. For example, if we have three classes, $y \in \{1, 2, 3\}$, we create copies of the original datasets and modify them. In the first copy, we replace all labels not equal to 1 by 0. In the second copy, we replace all labels not equal to 2 by 0. In the third copy, we replace all labels not equal to 3 by 0. Now we have three binary classification problems where we have to learn to distinguish between labels 1 and 0, 2 and 0, and between labels 3 and 0.

Once we have the three models and we need to classify the new input feature vector $\mathbf{x}$, we apply the three models to the input, and we get three predictions. We then pick the prediction of a non-zero class which is *the most certain*. Remember that in logistic regression, the model returns not a label but a score $(0, 1)$ that can be interpreted as the probability that the label is positive. We can also interpret this score as the certainty of prediction. In SVM, the analog of certainty is the distance from the input $\mathbf{x}$ to the decision boundary. This distance is given by,

$$d = \frac{\mathbf{w}^*\mathbf{x} + b^*}{\|w\|}.$$

The larger the distance, the more certain is the prediction. Most learning algorithm either can be naturally converted to a multiclass case, or they return a score we can use in the one versus rest strategy.

7.3 One-Class Classification

One-class classification, also known as *unary classification* or *class modeling*, tries to identify objects of a specific class among all objects, by learning from a training set containing only the objects of that class. That is different from and more difficult than the traditional classification problem, which tries to distinguish between two or more classes with the training set containing objects from all classes. A typical one-class classification problem is the classification of the traffic in a secure network as normal. In this scenario, there are few, if any, examples of the traffic under an attack or during an intrusion. However, the examples of normal traffic are often in abundance. One-class classification learning algorithms are used for outlier detection, anomaly detection, and novelty detection.

There are several one-class learning algorithms. The most widely used in practice are **one-class Gaussian**, **one-class kmeans**, **one-class kNN**, and **one-class SVM**.

The idea behind the one-class gaussian is that we model our data as if it came from a Gaussian distribution, more precisely *multivariate normal distribution* (MND). The probability density function (pdf) for MND is given by the following equation:

$$f_{\boldsymbol{\mu},\boldsymbol{\Sigma}}(\mathbf{x}) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)}{\sqrt{(2\pi)^{D}|\boldsymbol{\Sigma}|}} \quad ,$$

where $f_{\boldsymbol{\mu},\boldsymbol{\Sigma}}(\mathbf{x})$ returns the probability density corresponding to the input feature vector $\mathbf{x}$. Probability density can be interpreted as the likelihood that example $\mathbf{x}$ was drawn from the probability distribution we model as an MND. Values $\boldsymbol{\mu}$ (a vector) and $\boldsymbol{\Sigma}$ (a matrix) are the parameters we have to learn. The **maximum likelihood** criterion (similarly to how we solved the logistic regression learning problem) is optimized to find the optimal values for these two parameters. $|\boldsymbol{\Sigma}| = \det \boldsymbol{\Sigma}$ is the *determinant* of the matrix Σ; the notation $\mathbf{a}^{\mathrm{T}}$ means the *transpose* of the vector $\mathbf{a}$, and $\boldsymbol{\Sigma}^{-1}$ is the *inverse* of the matrix $\boldsymbol{\Sigma}$.

If the terms *determinant*, *transpose*, and *inverse* are new to you, don't worry. These are standard operations on vector and matrices from the branch of mathematics called *matrix theory.* If you feel the need to know what they are, Wikipedia explains these concepts very well.

In practice, the numbers in the vector $\boldsymbol{\mu}$ determine the place where the curve of our Gaussian distribution is centered, while the numbers in $\boldsymbol{\Sigma}$ determine the shape of the curve. For a training set consisting of two-dimensional feature vectors, an example of the one-class Gaussian model is given in fig 2.

Once we have our model parametrized by $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ learned from the data, we predict the likelihood of every input $\mathbf{x}$ by using $f_{\boldsymbol{\mu},\boldsymbol{\Sigma}}(\mathbf{x})$. Only if the likelihood is above a certain

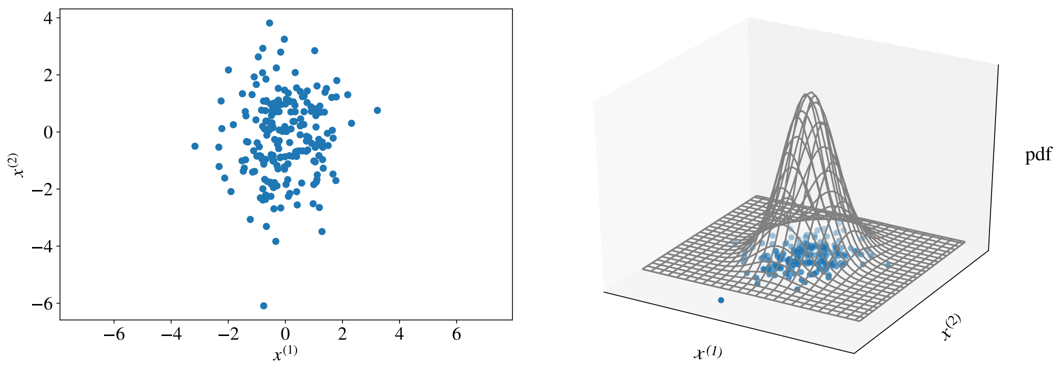

Figure 2: One-class classification solved using the one-class gaussian method. Left: two-dimensional feature vectors. Right: the MND curve that maximizes the likelihood of the examples on the left.

threshold, we predict that the example belongs to our class; otherwise, it is classified as the outlier. The value of the threshold is found experimentally or using an "educated guess."

When the data has a more complex shape, a more advanced algorithm can use a combination of several Gaussians (called a mixture of Gaussians). In this case, there are more parameters to learn from data: one $\boldsymbol{\mu}$ and one $\boldsymbol{\Sigma}$ for each Gaussian as well as the parameters that allow combining multiple Gaussians to form one pdf. In Chapter 9, we consider a mixture of Gaussians with an application to clustering.

One-class kmeans and one-class kNN are based on a similar principle as that of one-class Gaussian: build some model of the data and then define a threshold to decide whether our new feature vector looks similar to other examples according to the model. In the former, all training examples are clustered using the **kmeans** clustering algorithm and, when a new example $\mathbf{x}$ is observed, the distance $d(\mathbf{x})$ is calculated as the minimum distance between $\mathbf{x}$ and the center of each cluster. If $d(\mathbf{x})$ is less than a particular threshold, then $\mathbf{x}$ belongs to the class.

One-class SVM, depending on formulation, tries either 1) to separate all training examples from the origin (in the feature space) and maximize the distance from the hyperplane to the origin, or 2) to obtain a spherical boundary around the data by minimizing the volume of this hypersphere. I leave the description of the one-class kNN algorithm, as well as the details of the one-class kmeans and one-class SVM for the complementary reading.

Figure 3: A picture labeled as "people", "concert", and "nature".

7.4 Multi-Label Classification

In **multi-label classification**, each training example doesn't just have one label, but several of them. For instance, if we want to describe an image, we could assign several labels to it: "people," "concert," "nature," all three at the same time (fig. 3).

If the number of possible values for labels is high, but they are all of the same nature, like tags, we can transform each labeled example into several labeled examples, one per label. These new examples all have the same feature vector and only one label. That becomes a multiclass classification problem. We can solve it using the one versus rest strategy. The only difference with the usual multiclass problem is that now we have a new hyperparameter: threshold. If the prediction score for some label is above the threshold, this label is predicted for the input feature vector. In this scenario, multiple labels can be predicted for one feature vector. The value of the threshold is chosen using the validation set.

Analogously, algorithms that naturally can be made multiclass (decision trees, logistic regression and neural networks among others) can be applied to multi-label classification problems. Because they return the score for each class, we can define a threshold and then assign multiple labels to one feature vector if the threshold is above some value chosen experimentally using the validation set.

Neural networks algorithms can naturally train multi-label classification models by using the **binary cross-entropy** cost function. The output layer of the neural network, in this case, has one unit per label. Each unit of the output layer has the sigmoid activation function. Accordingly, each label l is binary ($y_{i,l} \in \{0, 1\}$), where $l = 1, \ldots, L$ and $i = 1, \ldots, N$. The binary cross-entropy of predicting the probability $\hat{y}_{i,l}$ that example $\mathbf{x}_i$ has label l is defined as $-(y_{i,l} \ln(\hat{y}_{i,l}) + (1 - y_{i,l}) \ln(1 - \hat{y}_{i,l}))$. The minimization criterion is simply the average of all binary cross-entropy terms across all training examples and all labels of those examples.

In cases where the number of possible values each label can take is small, one can convert multilabel into a multiclass problem using a different approach. Imagine the following problem. We want to label images and labels can be of two types. The first type of label can have two possible values: $\{photo, painting\}$; the label of the second type can have three possible values $\{portrait, paysage, other\}$. We can create a new fake class for each combination of the two original classes, like this:

Fake Class	Real Class 1	Real Class 2
1	photo	portrait
2	photo	paysage
3	photo	other
4	painting	portrait
5	painting	paysage
6	painting	other

Now we have the same labeled examples, but we replace real multi-labels with one fake label with values from 1 to 6. This approach works well in practice when there are not too many possible combinations of classes. Otherwise, you need to use much more training data to compensate for an increased set of classes.

The primary advantage of this latter approach is that you keep your labels correlated, contrary to the previously seen methods that predict each label independently of one another. Correlation between labels can be an essential property in many problems. For example, if you want to predict for an email message whether it's *spam* or *not_spam* at the same time as you predict whether it's *ordinary* or *priority* email. You would like to avoid predictions like $[spam, priority]$.

7.5 Ensemble Learning

Ensemble learning is a learning paradigm that, instead of trying to learn one super-accurate model, focuses on training a large number of low-accuracy models and then combining the predictions given by those weak models to obtain a high-accuracy **meta-model**.

Low-accuracy models are usually learned by **weak learners**, that is learning algorithms that cannot learn complex models, and thus are typically fast at the training and at the prediction time. The most frequently used weak learner is a decision tree learning algorithm in which we often stop splitting the training set after just a few iterations. The obtained trees are shallow and not particularly accurate, but the idea behind ensemble learning is that if the trees are not identical and each tree is at least slightly better than random guessing, then we can obtain high accuracy by combining a large number of such trees.

To obtain the prediction for input $\mathbf{x}$, the predictions of each weak model are combined using some sort of weighted voting. The specific form of vote weighting depends on the algorithm, but, independently of the algorithm, the idea is the same: if the council of weak models predicts that the message is spam, then we assign the label *spam* to $\mathbf{x}$.

Two most widely used and effective ensemble learning algorithms are **random forest** and **gradient boosting**.

7.5.1 Random Forest

There are two ensemble learning paradigms: **bagging** and **boosting**. Bagging consists of creating many "copies" of the training data (each copy is slightly different from another) and then apply the weak learner to each copy to obtain multiple weak models and then combine them. The bagging paradigm is behind the **random forest** learning algorithm.

The "vanilla" bagging algorithm works like follows. Given a training set, we create B random samples S_b (for each $b = 1, \ldots, B$) of the training set and build a decision tree model f_b using each sample S_b as the training set. To sample S_b for some b, we do the *sampling with replacement*. This means that we start with an empty set, and then pick at random an example from the training set and put its exact copy to S_b by keeping the original example in the original training set. We keep picking examples at random until the $|S_b| = N$.

After training, we have B decision trees. The prediction for a new example $\mathbf{x}$ is obtained as the average of B predictions:

$$y \leftarrow \hat{f}(\mathbf{x}) \stackrel{\text{def}}{=} \frac{1}{B} \sum_{b=1}^{B} f_b(\mathbf{x}),$$

in the case of regression, or by taking the majority vote in the case of classification.

The random forest algorithm is different from the vanilla bagging in just one way. It uses a modified tree learning algorithm that inspects, at each split in the learning process, a random subset of the features. The reason for doing this is to avoid the correlation of the trees: if one or a few features are very strong predictors for the target, these features will be selected to split examples in many trees. This would result in many correlated trees in our "forest." Correlated predictors cannot help in improving the accuracy of prediction. The main reason behind a better performance of model ensembling is that models that are good will likely agree on the same prediction, while bad models will likely disagree on different ones. Correlation will make bad models more likely to agree, which will hamper the majority vote or the average.

The most important hyperparameters to tune are the number of trees, B, and the size of the random subset of the features to consider at each split.

Random forest is one of the most widely used ensemble learning algorithms. Why is it so effective? The reason is that by using multiple samples of the original dataset, we reduce the **variance** of the final model. Remember that the low variance means low **overfitting**. Overfitting happens when our model tries to explain small variations in the dataset because our dataset is just a small sample of the population of all possible examples of the phenomenon we try to model. If we were unlucky with how our training set was sampled, then it could contain some undesirable (but unavoidable) artifacts: noise, outliers and over- or underrepresented examples. By creating multiple random samples with replacement of our training set, we reduce the effect of these artifacts.

7.5.2 Gradient Boosting

Another effective ensemble learning algorithm is gradient boosting. Let's first look at gradient boosting for regression. To build a strong regressor, we start with a constant model $f = f_0$ (just like we did in ID3):

$$f = f_0(\mathbf{x}) \stackrel{\text{def}}{=} \frac{1}{N}\sum_{i=1}^{N} y_i.$$

Then we modify labels of each example $i = 1, \ldots, N$ in our training set like follows:

$$\hat{y}_i \leftarrow y_i - f(\mathbf{x}_i), \tag{2}$$

where $\hat{y}_i$, called the *residual*, is the new label for example $\mathbf{x}_i$.

Now we use the modified training set, with residuals instead of original labels, to build a new decision tree model, f_1. The boosting model is now defined as $f \stackrel{\text{def}}{=} f_0 + \alpha f_1$, where α is the learning rate (a hyperparameter).

Then we recompute the residuals using eq. 2 and replace the labels in the training data once again, train the new decision tree model f_2, redefine the boosting model as $f \stackrel{\text{def}}{=} f_0 + \alpha f_1 + \alpha f_2$ and the process continues until the maximum of M (another hyperparameter) trees are combined.

Intuitively, what's happening here? By computing the residuals, we find how well (or poorly) the target of each training example is predicted by the current model f. We then train another tree to fix the errors of the current model (this is why we use residuals instead if real labels) and add this new tree to the existing model with some weight α. Therefore, each additional tree added to the model partially fixes the errors made by the previous trees until the maximum number of trees are combined.

Now you should reasonably ask why the algorithm is called *gradient* boosting? In gradient boosting, we don't calculate any gradient contrary to what we did in Chapter 4 for linear regression. To see the similarity between gradient boosting and gradient descent remember why we calculated the gradient in linear regression: we did that to get an idea on where we should move the values of our parameters so that the MSE cost function reaches its minimum. The gradient showed the direction, but we didn't know how far we should go in this direction, so we used a small step α at each iteration and then reevaluated our direction. The same happens in gradient boosting. However, instead of getting the gradient directly, we use its proxy in the form of residuals: they show us how the model has to be adjusted so that the error (the residual) is reduced.

The three principal hyperparameters to tune in gradient boosting are the number of trees, the learning rate, and the depth of trees — all three affect model accuracy. The depth of trees also affects the speed of training and prediction: the shorter, the faster.

It can be shown that training on residuals optimizes the overall model f for the mean squared error criterion. You can see the difference with bagging here: boosting reduces the bias (or underfitting) instead of the variance. As such, boosting can overfit. However, by tuning the depth and the number of trees, overfitting can be largely avoided.

The gradient boosting algorithm for classification is similar, but the steps are slightly different. Let's consider the binary case. Assume we have M regression decision trees. Similarly to logistic regression, the prediction of the ensemble of decision trees is modeled using the sigmoid function:

$$\Pr(y = 1|\mathbf{x}, f) \overset{\text{def}}{=} \frac{1}{1 + e^{-f(\mathbf{x})}},$$

where $f(\mathbf{x}) = \sum_{m=1}^{M} f_m(\mathbf{x})$ and f_m is a regression tree.

Again, like in logistic regression, we apply the maximum likelihood principle by trying to find such an f that maximizes $L_f = \sum_{i=1}^{N} \ln(\Pr(y_i = 1|\mathbf{x}_i, f))$. Again, to avoid numerical overflow, we maximize the sum of log-likelihoods rather than the product of likelihoods.

The algorithm starts with the initial constant model $f = f_0 = \frac{p}{1-p}$, where $p = \frac{1}{N}\sum_{i=1}^{N} y_i$. (It can be shown that such initialization is optimal for the sigmoid function.) Then at each iteration m, a new tree f_m is added to the model. To find the best f_m, first the partial derivative g_i of the current model is calculated for each $i = 1, \ldots, N$:

$$g_i = \frac{dL_f}{df},$$

where f is the ensemble classifier model built at the previous iteration $m - 1$. To calculate g_i we need to find the derivatives of $\ln(\Pr(y_i = 1|\mathbf{x}_i, f))$ with respect to f for all i. Notice that $\ln(\Pr(y_i = 1|\mathbf{x}_i, f)) \overset{\text{def}}{=} \ln(\frac{1}{1+e^{-f(\mathbf{x}_i)}})$. The derivative of the right-hand term in the previous equation with respect to f equals to $\frac{1}{e^{f(\mathbf{x}_i)}+1}$.

We then transform our training set by replacing the original label y_i with the corresponding partial derivative g_i, and we build a new tree f_m using the transformed training set. Then we find the optimal update step ρ_m as:

$$\rho_m = \arg\max_{\rho} L_{f+\rho f_m}.$$

At the end of iteration m, we update the ensemble model f by adding the new tree f_m:

$$f \leftarrow f + \alpha \rho_m f_m.$$

We iterate until $m = M$, then we stop and return the ensemble model f.

Gradient boosting is one of the most powerful machines learning algorithms. Not just because it creates very accurate models, but also because it is capable of handling huge datasets with millions of examples and features. It usually outperforms random forest in accuracy but, because of its sequential nature, can be significantly slower in training.

7.6 Learning to Label Sequences

A sequence is one the most frequently observed types of structured data. We communicate using sequences of words and sentences, we execute tasks in sequences, our genes, the music we listen and videos we watch, our observations of a continuous process, such as a moving car or the price of a stock are all sequential.

In sequence labeling, a labeled sequential example is a pair of lists (X, Y), where X is a list of feature vectors, one per time step, Y is a list of the same length of labels. For example, X could represent words in a sentence such as ["big", "beautiful", "car"], and Y would be the list of the corresponding parts of speech, such as ["adjective", "adjective", "noun"]). More formally, in an example i, $X_i = [\mathbf{x}_i^1, \mathbf{x}_i^2, \ldots, \mathbf{x}_i^{size_i}]$, where $size_i$ is the length of the sequence of the example i, $Y_i = [y_i^1, y_i^2, \ldots, y_i^{size_i}]$ and $y_i \in \{1, 2, \ldots, C\}$.

You have already seen that an RNN can be used to annotate a sequence. At each time step t, it reads an input feature vector $\mathbf{x}_i^{(t)}$, and the last recurrent layer outputs a label $y_{last}^{(t)}$ (in the case of binary labeling) or $\mathbf{y}_{last}^{(t)}$ (in the case of multiclass or multilabel labeling).

However, RNN is not the only possible model for sequence labeling. The model called **Conditional Random Fields** (CRF) is a very effective alternative that often performs well in practice for the feature vectors that have many informative features. For example, imagine we have the task of **named entity extraction** and we want to build a model that would label each word in the sentence such as "I go to San Francisco" with one of the following classes: $\{location, name, company\_name, other\}$. If our feature vectors (which represent words) contain such binary features as "whether or not the word starts with a capital letter" and "whether or not the word can be found in the list of locations," such features would be very informative and help to classify the words *San* and *Francisco* as *location*. Building handcrafted features is known to be a labor-intensive process that requires a significant level of domain expertise.

CRF is an interesting model and can be seen as a generalization of logistic regression to sequences. However, in practice, it has been outperformed by bidirectional deep gated RNN for sequence labeling tasks. CRFs are also significantly slower in training which makes them difficult to apply to large training sets (with hundreds of thousands of examples). Additionally, a large training set is where a deep neural network thrives.

7.7 Sequence-to-Sequence Learning

Sequence-to-sequence learning (often abbreviated as seq2seq learning) is a generalization of the sequence labeling problem. In seq2seq, X_i and Y_i can have different length. seq2seq models have found application in machine translation (where, for example, the input is an English sentence, and the output is the corresponding French sentence), conversational interfaces (where the input is a question typed by the user, and the output is the answer from the machine), text summarization, spelling correction, and many others.

Many but not most sequence-to-sequence learning problems are currently best solved by neural networks. Machine translation is a notorious example. There are multiple neural network architectures for seq2seq which perform better than others depending on the task. All those network architectures have one property in common: they have two parts, an **encoder** and a **decoder** (for this reason they are also known as **encoder-decoder** neural networks).

In seq2seq learning, the encoder is a neural network that accepts sequential input. It can be an RNN, but also a CNN or some other architecture. The role of the encoder is to read the input and generate some sort of state (similar to the state in RNN) that can be seen as a numerical representation of the *meaning* of the input the machine can work with. The meaning of some entity, whether it be an image, a text or a video, is usually a vector or a matrix that contains real numbers. This vector (or matrix) is called in the machine learning jargon the **embedding** of the input.

The decoder in seq2seq learning is another neural network that takes an embedding as input and is capable of generating a sequence of outputs. As you could have already guessed, that embedding comes from the encoder. To produce a sequence of outputs, the decoder takes a *start of sequence* input feature vector $\mathbf{x}^{(0)}$ (typically all zeroes), produces the first output $\mathbf{y}^{(1)}$, updates its state by combining the embedding and the input $\mathbf{x}^{(0)}$, and then uses the output $\mathbf{y}^{(1)}$ as its next input $\mathbf{x}^{(1)}$. For simplicity, the dimensionality of $\mathbf{y}^{(t)}$ can be the same as that of $\mathbf{x}^{(t)}$; however, it is not strictly necessary. As we saw in Chapter 6, each layer of an

RNN can produce many simultaneous outputs: one can be used to generate the label $\mathbf{y}^{(t)}$, while another one, of different dimensionality, can be used as the $\mathbf{x}^{(t)}$.

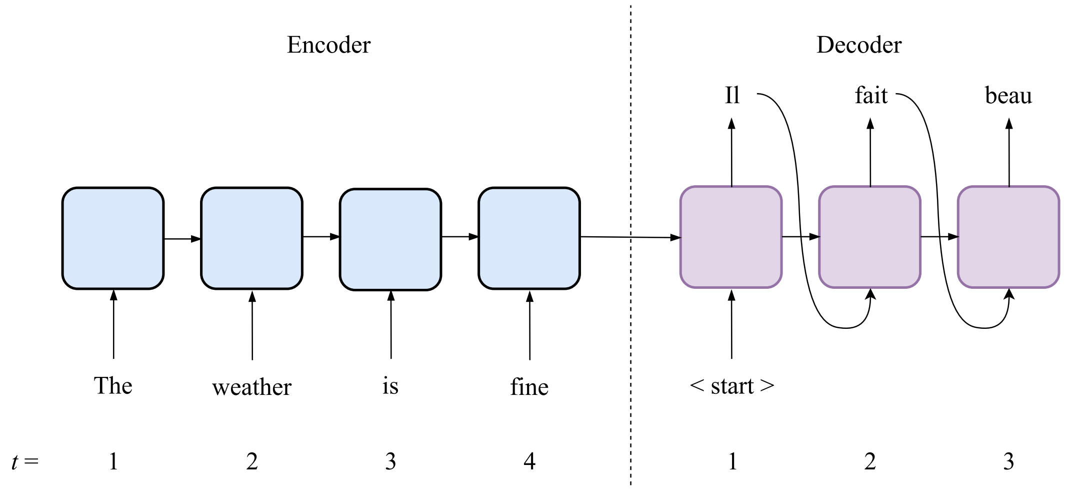

Figure 4: A traditional seq2seq architecture.

Both encoder and decoder are trained simultaneously using the training data. The errors at the decoder output are propagated to the encoder via backpropagation.

A traditional seq2seq architecture is illustrated in fig. 4. More accurate predictions can be obtained using an architecture with **attention**. Attention mechanism is implemented by an additional set of parameters that combine some information from the encoder (in RNNs, this information is the list of state vectors of the last recurrent layer from all encoder time steps) and the current state of the decoder to generate the label. That allows for even better retention of long-term dependencies than provided by gated units and bidirectional RNN. A seq2seq architecture with attention is illustrated in fig. 5.

Sequence-to-sequence learning is a relatively new research domain. Novel network architectures are regularly discovered and published. Training such architectures can be challenging as the number of hyperparameters to tune and other architectural decisions can be overwhelming. I recommend consulting the book's wiki for the state of the art material, tutorials and code samples.

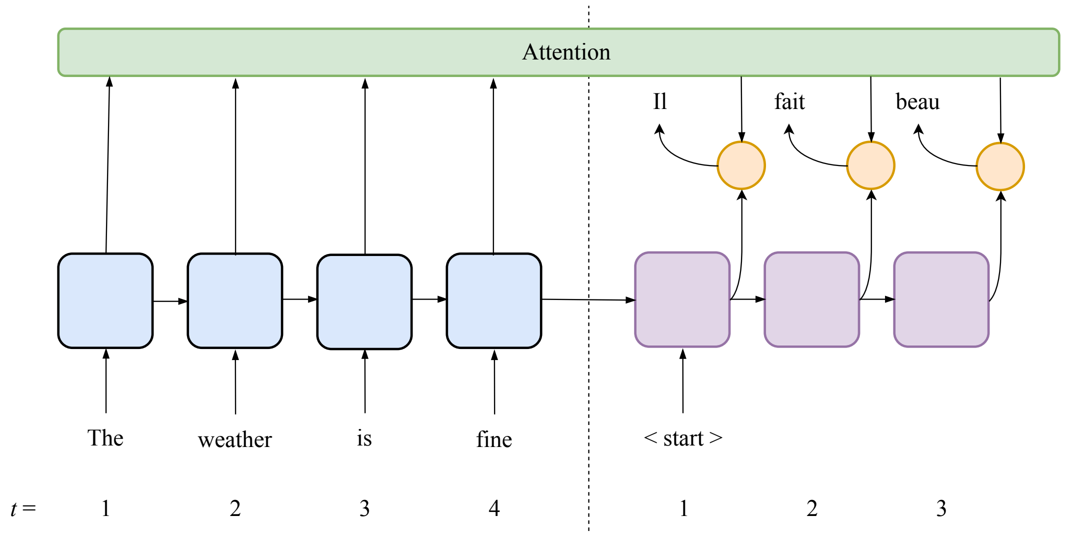

Figure 5: A seq2seq architecture with attention.

7.8 Active Learning

Active learning is an interesting supervised learning paradigm. It is usually applied when obtaining labeled examples is costly. That is often the case in the medical or financial domains, where the opinion of an expert may be required to annotate patients' or customers' data. The idea is that we start the learning with relatively few labeled examples, and a large number of unlabeled ones, and then add labels only to those examples that contribute the most to the model quality.

There are multiple strategies of active learning. Here, we discuss only the following two:

1) data density and uncertainty based, and
2) support vector-based.

The former strategy applies the current model f, trained using the existing labeled examples, to each of the remaining unlabelled examples (or, to save the computing time, to some random sample of them). For each unlabeled example $\mathbf{x}$, the following importance score is computed: $density(\mathbf{x}) \cdot uncertainty_f(\mathbf{x})$. Density reflects how many examples surround $\mathbf{x}$ in its close neighborhood, while $uncertainty_f(\mathbf{x})$ reflects how uncertain the prediction of the model f is for $\mathbf{x}$. In binary classification with sigmoid, the closer the prediction score is to 0.5, the more uncertain is the prediction. In SVM, the closer the example is to the decision boundary, the most uncertain is the prediction.

In multiclass classification, *entropy* can be used as a typical measure of uncertainty:

$$\mathrm{H}_f(\mathbf{x}) = -\sum_{c=1}^{C} \Pr(y^{(c)}; f(\mathbf{x})) \ln \Pr(y^{(c)}; f(\mathbf{x})),$$

where $\Pr(y^{(c)}; f(\mathbf{x}))$ is the probability score the model f assigns to class $y^{(c)}$ when classifying $\mathbf{x}$. You can see that if for each $y^{(c)}$, $f(y^{(c)}) = \frac{1}{C}$ then the model is the most uncertain and the entropy is at its maximum of 1; on the other hand, if for some $y^{(c)}$, $f(y^{(c)}) = 1$, then the model is certain about the class $y^{(c)}$ and the entropy is at its minimum of 0.

Density for the example $\mathbf{x}$ can be obtained by taking the average of the distance from $\mathbf{x}$ to each of its k nearest neighbors (with k being a hyperparameter).

Once we know the importance score of each unlabeled example, we pick the one with the highest importance score and ask the expert to annotate it. Then we add the new annotated example to the training set, rebuild the model and continue the process until some stopping criterion is satisfied. A stopping criterion can be chosen in advance (the maximum number of requests to the expert based on the available budget) or depend on how well our model performs according to some metric.

The support vector-based active learning strategy consists in building an SVM model using the labeled data. We then ask our expert to annotate the unlabeled example that lies the closest to the hyperplane that separates the two classes. The idea is that if the example lies closest to the hyperplane, then it is the least certain and would contribute the most to the reduction of possible places where the true (the one we look for) hyperplane could lie.

Some active learning strategies can incorporate the cost of asking an expert for a label. Others *learn* to ask expert's opinion. The "query by committee" strategy consists of training multiple models using different methods and then asking an expert to label example on which those models disagree the most. Some strategies try to select examples to label so that the variance or the bias of the model are reduced the most.

7.9 Semi-Supervised Learning

In **semi-supervised learning** (SSL) we also have labeled a small fraction of the dataset; most of the remaining examples are unlabeled. Our goal is to leverage a large number of unlabeled examples to improve the model performance without asking an expert for additional labeled examples.

Historically, there were multiple attempts at solving this problem. None of them could be called universally acclaimed and frequently used in practice. For example, one frequently cited SSL method is called "self-learning." In self-learning, we use a learning algorithm to build the initial model using the labeled examples. Then we apply the model to all unlabeled examples and label them using the model. If the confidence score of prediction for some unlabeled example $\mathbf{x}$ is higher than some threshold (chosen experimentally), then we add this labeled example to our training set, retrain the model and continue like this until a stopping criterion is satisfied. We could stop, for example, if the accuracy of the model has not been improved during the last m iterations.

The above method can bring some improvement to the model compared to just using the initially labeled dataset, but the increase in performance usually is not very impressive. Furthermore, in practice, the quality of the model could even decrease. That depends on the properties of the statistical distribution the data was drawn from, which we usually do not know.

On the other hand, the recent advancements in neural network learning brought some impressive results. For example, it was shown that for some datasets, such as MNIST (a frequent testbench in computer vision that consists of labeled images of handwritten digits from 0 to 9) the model trained in a semi-supervised way has an almost perfect performance with just 10 labeled examples per class (100 labeled examples overall). For comparison, MNIST contains 70,000 labeled examples (60,000 for training and 10,000 for test). The neural network architecture that attained such a remarkable performance is called a **ladder network**. To understand ladder networks you have to understand what an **autoencoder** is.

An autoencoder is a feed-forward neural network with an encoder-decoder architecture. It is trained to reconstruct its input. So the training example is a pair $(\mathbf{x}, \mathbf{x})$. We want the output $\hat{\mathbf{x}}$ of the model $f(\mathbf{x})$ to be as similar to the input $\mathbf{x}$ as possible.

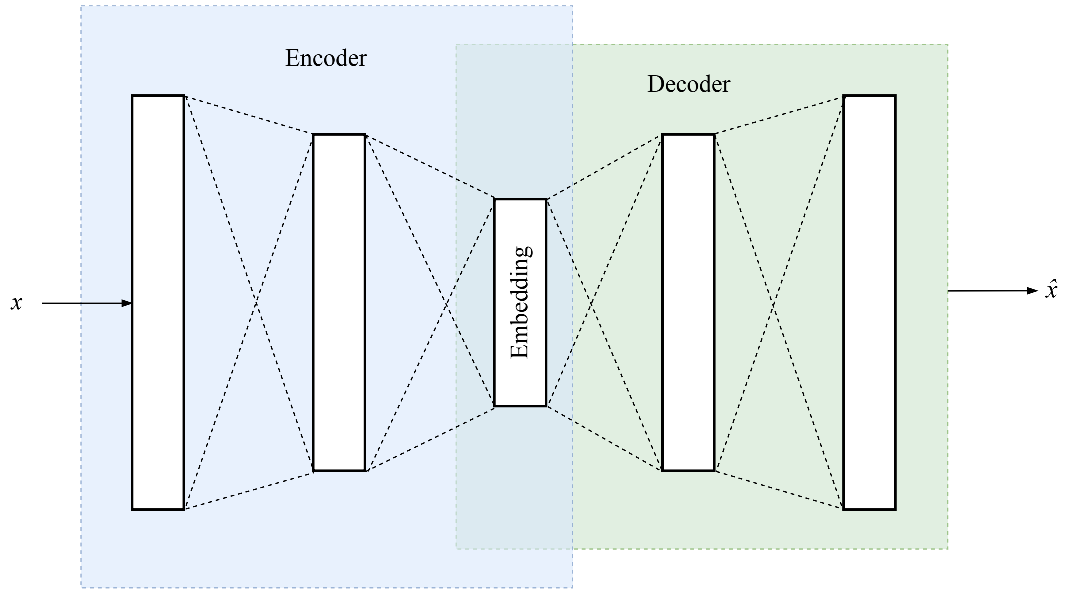

Figure 6: Autoencoder.

An important detail here is that an autoencoder's network looks like an hourglass with a **bottleneck layer** in the middle that contains the embedding of the D-dimensional input vector; the embedding layer usually has much fewer units than D. The goal of the decoder is to reconstruct the input feature vector from this embedding. Theoretically, it is sufficient to have 10 units in the bottleneck layer to successfully encode MNIST images. In a typical autoencoder schematically depicted in fig. 6, the cost function is usually either the mean squared error (when features can be any number) or the negative log-likelihood (when features are binary and the units of the last layer of the decoder have the sigmoid activation function). If the cost is the mean squared error, then it is given by:

$$\frac{1}{N}\sum_{i=1}^{N}\|\mathbf{x}_i - f(\mathbf{x}_i)\|^2,$$

where $\|\mathbf{x}_i - f(\mathbf{x}_i)\|$ is the Euclidean distance between two vectors.

A **denoising autoencoder** corrupts the left-hand side $\mathbf{x}$ in the training example $(\mathbf{x}, \mathbf{x})$ by adding some random perturbation to the features. If our examples are grayscale images with pixels represented as values between 0 and 1, usually a **normal Gaussian noise** is added to each feature. For each feature j of the input feature vector $\mathbf{x}$ the noise value $n^{(j)}$ is sampled from the following distribution:

$$n^{(j)} \sim \frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{(-\mu)^2}{2\sigma^2}\right),$$

where the notation $\sim$ means "sampled from," π is the constant $3.14159\ldots$ and μ is a hyperparameter that has to be tuned. The new, corrupted value of the feature $x^{(j)}$ is given by $x^{(j)} + n^{(j)}$.

A ladder network is a denoising autoencoder with an upgrade. The encoder and the decoder have the same number of layers. The bottleneck layer is used directly to predict the label (using the softmax activation function). The network has several cost functions. For each layer l of the encoder and the corresponding layer l of the decoder, one cost C_d^l penalizes the difference between the outputs of the two layers (using the squared Euclidean distance). When a labeled example is used during training, another cost function, C_c, penalizes the error in prediction of the label (the negative log-likelihood cost function is used). The combined cost function, $C_c + \sum_{l=1}^{L}\lambda_l C_d^l$ (averaged over all examples in the batch), is optimized by the stochastic gradient descent with backpropagation. The hyperparameters λ_l for each layer l determine the tradeoff between the classification and encoding-decoding cost.

In the ladder network, not just the input is corrupted with the noise, but also the output of each encoder layer (during training). When we apply the trained model to the new input $\mathbf{x}$ to predict its label, we do not corrupt the input.

Other semi-supervised learning techniques, not related to training neural networks, exist. One of them implies building the model using the labeled data and then cluster the unlabeled and labeled examples together using any clustering technique (we consider some of them in Chapter 9).

For each new example, we then output as a prediction the majority label in the cluster it belongs to. Another technique, called S3VM, is based on using SVM. We build one SVM model for each possible labeling of unlabeled examples and then we pick the model with the largest margin. The paper on S3VM describes an approach that allows solving this problem without actually enumerating all possible labelings.

7.10 One-Shot Learning

This chapter would be incomplete without mentioning two other important supervised learning paradigms. One of them is **one-shot learning**. In one-shot learning, typically applied in face recognition, we want to build a model that can recognize that two photos of the same person represent that same person. If we present to the model two photos of two different people, we expect the model to recognize that the two people are different.

One way to build such a model is to train a **siamese neural network** (SNN). An SNN can be implemented as any kind of neural network, a CNN, an RNN, or an MLP. What matters is how we train the network.

To train an SNN, we use the **triplet loss** function. For example, let us have three images of a face: the image A (for anchor), the image P (for positive) and the image N (for negative). A and P are two different pictures of the same person; N is a picture of another person. Each training example i is now a triplet (A_i, P_i, N_i).

Let's say we have a neural network model f that can take a picture of a face as input and output an embedding of this picture. The triplet loss for one example is defined as,

$$\max(\|f(A_i) - f(P_i)\|^2 - \|f(A_i) - f(N_i)\|^2 + \alpha, 0). \tag{3}$$

The cost function is defined as the average triplet loss:

$$\frac{1}{N}\sum_{i=1}^{N}\max(\|f(A_i) - f(P_i)\|^2 - \|f(A_i) - f(N_i)\|^2 + \alpha, 0),$$

where α is a positive hyperparameter. Intuitively, $\|f(A) - f(P)\|^2$ is low when our neural network outputs similar embedding vectors for A and P; $\|f(A_i) - f(N_i)\|^2$ is high when the embedding for pictures of two different people are different. If our model works the way we want, then the term $m = \|f(A_i) - f(P_i)\|^2 - \|f(A_i) - f(N_i)\|^2$ will always be negative, because we subtract a high value from a small value. By setting α higher, we force the term

m to be even smaller, to make sure that the model learned to recognize the two same faces and two different faces with a high margin. If m is not small enough, then because of α the cost will be positive, and the model parameters will be adjusted in backpropagation.

Rather than randomly choose an image for N, a better way to create triplets for training is to use the current model after several epochs of learning and find candidates for N that are similar to A and P according to that model. Using random examples as N would significantly slow down the training because the neural network will easily see the difference between pictures of two random people, so the average triplet loss will be low most of the time and the parameters will not be updated fast enough.

To build an SNN, we first decide on the architecture of our neural network. For example, CNN is a typical choice if our inputs are images. Given an example, to calculate the average triplet loss, we apply, consecutively, the model to A, then to P, then to N, and then we compute the loss for that example using eq. 3. We repeat that for all triplets in the batch and then compute the cost; gradient descent with backpropagation propagates the cost through the network to update its parameters.

It's a common misconception that for one-shot learning we need only one example of each entity for training. In practice, we need much more than one example of each person for the person identification model to be accurate. It's called one-shot because of the most frequent application of such a model: face-based authentication. For example, such a model could be used to unlock your phone. If your model is good, then you only need to have *one picture* of you on your phone and it will recognize you, and also it will recognize that someone else is not you. When we have the model, to decide whether two pictures A and $\hat{A}$ belong to the same person, we check if $\|f(A) - f(\hat{A})\|^2$ is less than some threshold τ, which is another hyperparameter of the model.

7.11 Zero-Shot Learning

We finish this chapter with **zero-shot learning**. It is a relatively new research area, so there are no algorithms that proved to have a significant practical utility yet. Therefore, I only outline here the basic idea and leave the details of various algorithms for further reading. In zero-shot learning (ZSL) we want to train a model to assign labels to objects. The most frequent application is to learn to assign labels to images.

However, we want the model to be able to predict labels that we didn't have in the training data. How is that possible?

The trick is to use embeddings not just to represent the input $\mathbf{x}$ but also to represent the output y. Imagine that we have a model that for any word in English can generate an embedding vector with the following property: if a word y_i has a similar meaning to the word y_k, then the embedding vectors for these two words will be similar. For example, if y_i is *Paris* and y_k is *Rome*, then they will have embeddings that are similar; on the other hand, if y_k is *potato*, then the embeddings of y_i and y_k will be dissimilar. Such embedding vectors are called "word embeddings," and they are usually compared using cosine similarity metrics[1].

Word embeddings have such a property that each dimension of the embedding represents a specific feature of the meaning of the word. For example, if our word embedding has four dimensions (usually they are much wider, between 50 and 300 dimensions), then these four dimensions could represent such features of the meaning as *animalness*, *abstractness*, *sourness*, and *yellowness* (yes, sounds funny, but it's just an example). So the word *bee* would have an embedding like this $[1, 0, 0, 1]$, the word *yellow* like this $[0, 1, 0, 1]$, the word *unicorn* like this $[1, 1, 0, 0]$. The values for each embedding are obtained using a specific training procedure applied to a vast text corpus.

Now, in our classification problem, we can replace the label y_i for each example i in our training set with its word embedding and train a multi-label model that predicts word embeddings. To get the label for a new example $\mathbf{x}$, we apply our model f to $\mathbf{x}$, get the embedding $\hat{\mathbf{y}}$ and then search among all English words those whose embeddings are the most similar to $\hat{\mathbf{y}}$ using cosine similarity.

Why does that work? Take a zebra for example. It is white, it is a mammal, and it has stripes. Take a clownfish: it is orange, not a mammal, and has stripes. Now take a tiger: it is orange, it has stripes, and it is a mammal. If these three features are present in word embeddings, the CNN would learn to detect these same features in pictures. Even if the label *tiger* was absent in the training data, but other objects including zebras and clownfish were, then the CNN will most likely learn the notion of *mammalness*, *orangeness*, and *stripeness* to predict labels of those objects. Once we present the picture of a tiger to the model, those features will be correctly identified from the image and most likely the closest word embedding from our English dictionary to the predicted embedding will be that of *tiger*.

[1]I will show in Chapter 10 how to learn words embeddings from data.