My ML studies

Babak Poursartip

January 14, 2025

Contents

Chapter 1

Introduction to ML

1.1 General

- Supervised learning (SL): functional approximation from labeled data.
- Unsupervised learning (UL): functional description. Learning from unlabeled data.
- Reinforcement learning (RL): Learning from delayed reward.

Chapter 2

Supervised learning (SL)

There are two types of supervised learning:

- classification: taking some inputs and mapping it to some discrete labels. (true of false; male or female; SUV or sedan or truck; class1, class2, class3.)
- regression: returns continuous valued function. Mapping from input to some **real number**. Something like age is more like discrete number, so, it is also classification.

2.1 Terms

- Instances: Inputs, such as pictures, pixels, etc.
- Concept: A set of *functions* that maps inputs to outputs.
- Target concept: The actual function that can map inputs to outputs. This is the actual answer.
- Hypothesis class: Set of all the functions that we are going to think about.
- Sample (Training set): A set of instances with correct labels.
- Candidate: The best approximation of the target concept.

- Testing set: A set of instances with correct labels that was not visible to the learning algorithm during the training phase. It's used to determine the algorithm performance on novel data. we can argue that we learned something by memorization, but indeed, we just memorized the concepts. What we want is generalization.

Remark:

There are generally two ways a regression algorithm can make predictions: either by transforming the features or by transforming the target. Feature-transforming algorithms learn some mathematical function that takes features as an input and then combines and transforms them to produce an output that matches the target values in the training set. Linear regression and neural nets are of this kind.

Target-transforming algorithms use the features to group the target values in the training set and make predictions by averaging values in a group; a set of feature just indicates which group to average. Decision trees and nearest neighbors are of this kind.

The important thing is this: feature transformers generally can extrapolate target values beyond the training set given appropriate features as inputs, but the predictions of target transformers will always be bound within the range of the training set. If the time dummy continues counting time steps, linear regression continues drawing the trend line. Given the same time dummy, a decision tree will predict the trend indicated by the last step of the training data into the future forever. Decision trees cannot extrapolate trends. Random forests and gradient boosted decision trees (like XGBoost) are ensembles of decision trees, so they also cannot extrapolate trends.

There are two types of data for supervised learning: Structured and Unstructured data. The structured data is the data bases of data tabulated in tables. The Unstructured data refers to image, text, or audio. Generally, it is more complex to train a model with unstructured data.

2.2 SL: Decision tree

2.2.1 Introduction

Decision tree is a classification learning in the form of a sequence of decisions based on the attributes/features/questions (which forms the nodes) applied to every instance to assign it to a specific class. Answers to the questions

would be the edges of the trees. For example, deciding to eat at a restaurant or not, or classification of vehicles into sedans, SUVs, trucks, etc.

2.2.2 Representation of the decision tree:

A decision tree is a structure of nodes, edges and leaves that can be used to *represent* the data. We always start at the root of the tree, otherwise we don't look at any other attribute in the tree.

- Nodes represent attributes where you ask a question about it. (vehicle length, vehicle height, number of doors, etc.).
- Edges represent values/answers, a specific value for each attribute/question.
- Leaves represent the output (or final answer). For example, vehicle's type.

2.2.3 Algorithm to build a decision tree

Algorithm means how to create the decision tree. A decision tree *algorithm* is a sequence of steps that will lead you to the desired output. To form a decision tree, we need to come up with the attributes that can split the space, roughly in half.

- Pick the best attribute (the one that can split the data roughly in half). If this attribute added no valuable information (not a good split), it might cause overfitting.
- Ask a question about this attribute.
- Follow the correct answer path.
- Loop back to (1) till you narrow down the possibilities to one answer (the output).

Note that the usefulness of a question depends upon the answers we got to previous questions.

Purity: means all the samples are of the same kind. This is the ideal case for the leaf nodes. We have impurity if it is still a mix of various categories.

2.2.4 Expressiveness

- Decision trees can basically express any function using the input attributes.
- For Boolean functions (AND, OR, XOR, etc.), the truth table row will be translated into a path to a leaf.
- How many decision trees we can generate for a specific function/problem? For n boolean attributes with boolean output, the decision tree hypothesis space is very huge ($2^{(2^n)}$). (Babak note: this number of decision tree is not really distinct, see my notebook). This is why we need to design algorithms to efficiently search the hypothesis space for the best decision tree.

Decision trees with AND and OR are linear (they need linear number of nodes-ANY type problems). But XOR is a hard problem (requires 2^n nodes-PARITY type of problems).

2.2.5 ID3 algorithm to create a decision tree

We need to find the best attributes for each node, to narrow down the space with each question. ID3 (Inducing Decision Tree (3)) builds decision trees using a top-down, greedy approach. The greedy part of the approach comes from the fact that it will decide which attribute should be at the root of the tree by looking just one move ahead. It compares all available attributes to find which one classifies the data the best, but it doesn't look ahead (or behind) at other attributes to see which combinations of them classify the data the best. ID3 algorithm returns an optimal decision tree, but as the size of the training data and the number of attributes increase, it becomes more likely that running ID3 on it will return a suboptimal decision tree.

To train a decision tree, answer these questions:

- How to find the best attribute/feature for the decision tree at each level?
- When do you stop splitting?

Pseudo code:

1. Pick the best attribute A (the definitions of best attribute comes later). To select this attribute, a statistical test is used to determine for each attribute how well it alone classifies the training examples.

2. Split the data into subsets that correspond to the possible values of the best attribute.

3. Assign A as a decision attribute for a node.

4. For each value of A, create a descendant node.

5. Sort training examples to leaves.

6. Stopping criteria (see down there): f examples perfectly classified (means all data point are of the same class) or there is no more attributes – Stop splitting – else –Iterate over leaves

How to find the best attribute/feature for the decision tree at each level:
There are a couple of options. The most common method is called information gain: a measure that expresses how well an attribute splits the data into groups based on classification (maximize purity or minimize impurity). For example, an attribution returns *low information gain*, if it divides samples to two classes with an even yes and no, but the information gain is high, if each class has more of yeses or nos.

It quantifies the reduction in randomness (Entropy) over the labels we have with a set of data, based upon knowing the value of a particular attribute. A mathematical way to measure the gain, is entropy. It measures the homogeneity of a data set S's classifications. *Entropy ranges from 0, which means that all of the classifications in the data set are the same (either yes or no), to log_2 of the number of different classifications, which means that the classifications are equally distributed within the data set.* For a binary classification the entropy is only $log_2 1 = 1$ (if yes and no samples are equally divided). entropy is calculated as follows:

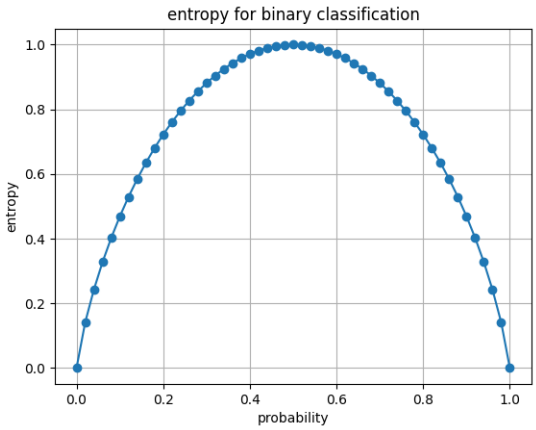
$$Entropy(S) = \sum_{i=1}^{c} -p_i \, log_2(p_i) \tag{2.1}$$

where:

- c corresponds to the number of different classifications (either for the attribute or the final output)
- p_i corresponds to the proportion of the data with the classification i

Here is the plot of entropy for a binary classifier, as the proportional of yes and no samples change:

Figure 2.1: Entropy



Information gain measures the reduction in entropy that results from partitioning the data on an attribute A, which is another way of saying that it represents how effective an attribute is at classifying the data. Given a set of training data S and an attribute A, the formula for information gain is:

$$Gain(S, A) = Entropy(S) - \sum_{v} \frac{|S_v|}{|S|} Entropy(S_v) \tag{2.2}$$

where v is all the possible values of attribute the attribute (for example, sunny, cloudy, rainy), and S_v is the total number of samples corresponding to this value of attribute (for example sunny).

For *if I play tennis game* (yes and no is the final output) with attributes weather (sunny, cloudy, rainy), temperature (hot, cold, mild), humidity (normal, high), first, we calculate the entropy of the entire set (Entropy(S)) (wrt the the final output classification, yes and no). For each attribute, we calculate the the entropy corresponding to each value of entropy, then we calculate the final entropy.

We want to maximize information gain, so we want the entropies of the partitioned data to be as low as possible, which explains why attributes that exhibit high information gain split training data into relatively heterogeneous groups.

As the size of the training data and the number of attributes increase, it becomes likelier that running ID3 on it will return a sub-optimal decision tree.

The other option we can use instead of Entropy is the **Gini function.**

When do you stop splitting?

- when a node is 100% pure
- when splitting a node will result in the tree exceeding a maximum depth (to avoid over-fitting)
- when improvement in purity score (information gain) are below a threshold
- when the number of examples in a node is below a threshold

2.2.6 Inductive bias

Two types of bias for classifiers:

- Restriction Bias: Describes the hypothesis set space (H) that we will consider for learning the tree. Complete hypothesis space (low Restriction Bias). Instead of looking at infinitely many functions, we only consider those that can be represented by the decision tree.
- Preference Bias: Describes the subset of hypothesis n ($n \in H$), from the hypothesis set space H, that we prefer.

Inductive bias of ID3 algorithm:

- it prefers the decision tree with good splits at top (even if a bad split generates the same outcome). This is because ID3 is greedy using info gain.
- it prefers correct outcome over incorrect because ID3 repeats until the labels are correctly classified.

- it prefers shorter trees (comes from the fact that we use good splits from the top).

2.2.7 Extending ID3 algorithm-other consideration

For continuous values attributes/features, we can classify the attributes based on thresholds (that exists in the data set). Without threshhold, the continuous data attribute provides the most information gain, while giving a decision tree that is not generalize well. To find the threshold, we can plot the values and decide based on a threshold the returns the best outcome, by calculating the information gain values for each threshold. One way to select the threshold is to take all the values that are mid points between the sorted list of training. For example, for 10 training samples, we need to test the info gain for 9 values, for this specific feature. We can even have multiple threshold for one feature (by creating multiple features).

Does it make sense to repeat an attribute along a path in a decision tree? No, it does not make sense for discrete values, but for continuous values, it makes sense, because indeed we are asking a different question (for a different range).

If the data for some attributes is missing, see 6.10.

Over-fitting + When do we stop? When everything classified correctly or if there is no more attribute. What if we have noise in data (different answer for the same instance)? Causes an infinite loop. What would be the stopping criterion, then? We want generalization, and we should avoid **over-fitting**. If the tree is too big/complicated, there is a chance that it over-fits. There are two popular approaches to avoid over-fitting in decision trees: stop growing the tree before it becomes too large or prune the tree after it becomes too large. Typically, a limit to a decision tree's growth will be specified in terms of the maximum number of layers, or depth, it's allowed to have.

One option to avoid over-fitting is that we can grow the trees, and use cross-validation to prevent over-fitting. The data available to train the decision tree will be split into a training set and validation/test set and we keep growing the tree with various maximum depths will be created based on the training set and then test it against the test set. The best will be selected (when the error grows, we stop growing the tree).

Pruning the tree, on the other hand, involves testing the original tree against pruned versions of it. Leaf nodes are taken away from the tree as long as the pruned tree performs better against test data than the larger

tree.

One-hot encoding: If there are multiple categories for a feature, we can use one-hot encoding to convert each category to a binary feature. Thus, jf a feature can take k values, one-hot encoding creates k binary features. There are libraries in the sklearn library for this task. See the details in the 6.1.

Regression: how to adapt decision trees for regression type of problems (continuous outputs)? Decision trees as we've defined them here don't transfer directly to regression problems. We no longer have a useful notion of information gain, so our approach at attribute sorting falls through. Instead, we can rely on purely statistical methods (like variance and correlation) to determine how important an attribute is. For leaves, too, we can do averages, local linear fit, or a host of other approaches that mathematically generalize with no regard for the meaning of the data.

2.2.8 Regression with Decision tree

We can use decision trees for classification (DecisionTreeClassifier) or for regression (DecisionTreeRegressor). Training a regressor is not any different from the classifier, but for the leaf node, we use the average of the continuous values of all the samples in that node.

The major difference here is that how you choose to split the data at the node. Instead of maximizing the entropy, we try to minimize the variance of the values for the continuous feature. Thus, we calculate the variance for the values at each leaf separated by the feature, then we use a weighted method to combine it.

$$\begin{gathered}\text{minimize the variance (means choose the feature with largest value)} = \\ \text{variance of all samples at the root} - \\ \Bigg[\\ \frac{\text{num of samples in the left leaf}}{\text{num of total samples}} \times \text{variance of left leaf} + \\ \frac{\text{num of samples in the right leaf}}{\text{num of total samples}} \times \text{variance of right leaf} \\ \Bigg]\end{gathered} \tag{2.3}$$

For example, if a feature divides 10 samples into two leaf nodes with 4

and 6 samples in each leaf node, then, the weighted average of variance is $\frac{4}{10} \times$ variance of left leaf $+ \frac{6}{10} \times$ variance of right leaf.

2.2.9 Changing the information gain formula is another option

The information gain formula used by the ID3 algorithm treats all of the variables the same, regardless of their distribution and their importance. This is a problem when it comes to continuous variables or discrete variables with many possible values because training examples may be few and far between for each possible value, which leads to low entropy and high information gain by virtue of splitting the data into small subsets, but results in a decision tree that might not generalize well.

One successful approach to deal with this is using a formula called GainRatio in the place of information gain. GainRatio tries to correct for information gain's natural bias toward attributes with many possible values by adding a denominator to information gain called SplitInformation. SplitInformation attempts to measure how broadly partitioned an attribute is and how evenly distributed those partitions are. In general, the SplitInformation of an attribute with n equally distributed values is $log_2 n$. These relatively large denominators significantly affect an attribute's chances of being the best attribute after an iteration of the ID3 algorithm and help to avoid choices that perform particularly well on the training data but not so well outside of it.

$$GainRatio(S, A) = \frac{Gain(S, A)}{SplitInformation(S, A)}$$

$$SplitInformation(S, A) = -\sum_{i=1}^{c} \frac{|S_i|}{S} \ log_2 \frac{|S_i|}{|S|}$$

2.2.10 Advantages and Disadvantages of Decision Trees (copied from a text)

Advantages:

- Simple to understand and to interpret.

- Requires little data preparation. Other techniques often require data normalization, dummy variables need to be created and blank values to be removed.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data. Other techniques are usually specialized in analyzing datasets that have only one type of variable.
- Able to handle multi-output problems.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by Boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
- Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

Disadvantages:

- Can create over-complex trees that do not generalize the data well. This is called overfitting. Mechanisms such as pruning (setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree) are necessary to avoid this problem.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble (random forest/XGBoost).
- The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot

guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.

- There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

2.2.11 When to use a decision tree

Decision trees and tree ensembles:

- Works well on tabular/structured data
- Not recommended for unstructured data such as text, image, audio. Use NN here.
- Decision tree is fast in training.
- Small decision trees may be human interpretable.

Neural Networks

- Works well on all types of data, including tabular/structured and unstructured data.
- May be slower than the decision tree for training.
- Works with transfer learning.
- When building a system of multiple models working together, it might be easier to string together multiple neural networks.

2.2.12 Random forest (Tree ensembles)

A single decision tree can be highly sensitive to small changes in the data. The random forest uses many trees, and it makes a prediction by averaging the predictions of each component tree. It generally has much better

predictive accuracy than a single decision tree and it works well with default parameters. If you keep modeling, you can learn more models with even better performance, but many of those are sensitive to getting the right parameters.

To build a tree ensemble (random forest), we use a technique called *Sampling With Replacement*. We are going to construct random sets of the same size of the original set, that are slightly different from the original set, by randomly selecting instances from the set (so, some of the training instances might be repeated in each set). Then, we train a decision tree using this data set. We continue this process to build B decision trees (Bagged decision tree). To randomize the feature choice further, at each node, when choosing a feature to use to split, if n features are available, pick a random subset of $k < n$ features and allow the algorithm to only choose from that subset of features. If n is large $\sqrt{n}$ is valid.

Pseudo code:
Given training set of size m
For b=1 to B:

- use sampling with replacement to create a new training set of size m
- train a decision tree on the new dataset

2.2.13 Boosted tree (XGBoost)

This is the most commonly used method of decision tree on samples. It requires a modification to the Bagged decision tree from the previous section 2.2.12. Basically, it is focusing on the part the data that the tree is not doing well.

Pseudo code:
Given training set of size m
For b=1 to B:

- use sampling with replacement to create a new training set of size m
- Instead of picking from all examples with equal $1/m$ probability, make it more likely to pick misclassified examples from previously trained trees.
- train a decision tree on the new dataset

2.3 SL: Regression and classification

Regression is the mapping of continuous inputs to continuous outputs, as opposed to the classification where we had mapping from discrete input to discrete output. Regression is similar to function approximation. Back then, regression meant falling back to the average/mean, but now, we mean, using functional form to approximate a bunch of points.

We use regression because the data itself is not accurate and has noise. If the data is exact, we need to use interpolation and spline methods, to exactly fit the line with the data.

2.3.1 Linear regression

Finding a linear function/relationship between the input data and the output. It might not be a good fit though. We want to find a line (linear function) such that it minimizes the deviation, or to be more precise, the squared error between the data points and the line. This is least square regression. Basically, this is fitting data with a curve, so that we can predict the results based on the model. Here we do not try to intersect every point, rather the curve is designed to follow the pattern of points. (The other approach is interpolation that we try to pass the line/curve through each data point.)

In summary, this is an approximate function that fits the shape or general trend of the data w/o necessarily matching the individual points. To find the line:

$$\begin{aligned} \text{data points: } & \{(x_i, y_i) : i = 1...n\} \\ \text{find a and b for the linear fit: } & y_i = ax_i + b \\ \text{such that minimizes error: } & e = \sum_{i=1}^{n} (y_i - (ax_i + b))^2 \end{aligned} \tag{2.4}$$

To find the the two unknowns of the linear fit, a and b, take the derivative of the error with respect to a and b and set it zero. That would be a system of equations with two unknowns. After solving the equation you have the fit. To have a nonlinear fit, we can use a polynomial model, as discussed in the next section.

The parameters of the model are W and b, but hyperparameters are learning rate, number of iterations, number of hidden layers, number of hidden

units in each layer, the choice of activation functions, momentum, mini batch size, and regularization parameters.

To do linear regression, we can use sklearn.linear_model.LinearRegression.

2.3.2 Non-linear/polynomial regression

$$f(x_i) = a_0 + a_1 x_i + a_2 x_i^2 + ... \tag{2.5}$$

The unknowns are $a_0, a_1, ...$ See my class notes for more details. To better fit the regression model, we can use higher order polynomial, but it leads to over-fitting.

2.3.3 Error

Training data has errors due to various reasons such as reading errors, sensor errors, transcription error, maliciously given data, etc. This is the reason we are using the regression not the interpolation.

To calculate the parameters of the model, we need to form the cost function (see 6.5.1) and minimize the error using an optimization technique, such as gradient descent 6.5.2.

2.3.4 Multiple Linear regression (multiple features)

For the regression, we can have multiple independent variables/features for the regression $(x_1, ..., x_n)$.

2.3.5 Other cases

Using discrete numbers is also possible for regression, however this is no trivial. We need to encode the attributes. For example, by enumerating the categories (giving a number to each possible value of this feature). It is a bit misleading because a correlation between the ordering of the enumeration and the its value can be interpolated. The other option is represent the value of the feature as a Boolean.

2.3.6 Classification

The output variable y would be a small handful of cases (as opposed to the linear regression with infinite range of numbers). Examples of (binary) clas-

sifications, include email spam detection, tumor detection, etc. One method is to use linear regression and map all predictions greater than 0.5 as a 1 and all less than 0.5 as a 0. This method doesn't work well because classification is not actually a linear function. Instead we need logistic regression. Logistic/Sigmoid function is (output values between 0 and 1, which is the probability of each category/class):

$$g(z) = \frac{1}{1+e^{-z}} \quad \text{where} \quad 0 < g(z) < 1 \tag{2.6}$$

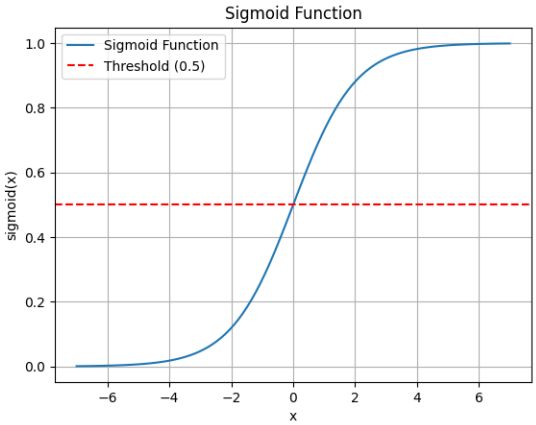


Figure 2.2: Sigmoid

In the language of linear regression, it means we feed the sigmoid func with the output of linear regression:

$$f_{\vec{w},b}(\vec{x}) = g(\vec{w}.\vec{x} + b) \tag{2.7}$$

In order to get our discrete 0 or 1 classification, we can define a *decision boundary*, such as 0.5, so, if $g(z) > 0.5 \rightarrow y = 1$, this means $z = \vec{w}.\vec{x} + b > 0$. By setting $\vec{w}.\vec{x} + b = 0$, we can find/plot the decision boundary.

2.4 SL: Neural network

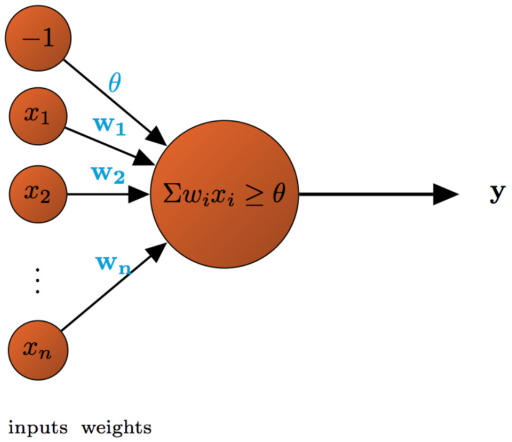
Inference is getting the prediction results from a trained model using forward propagation.

2.4.1 Perceptron

Perceptron is the simplest model of a brain cell. The Perceptron calculates the sum of products of the inputs X and their corresponding weights W, and then compare the result with an activation threshold. If the sum is greater than or equal the firing threshold θ, the perceptron returns one, otherwise, it is zero.

$$\sum_{i=1}^{k} X_i W_i \geq \theta \Rightarrow y = 1 \text{ otherwise } y = 0 \tag{2.8}$$

Figure 2.3: Perceptron



Layer: a group of neurons, which take the same input/features as input. A layer can have multiple or single neurons. We can assume the features as the input layer ($\overrightarrow{x}$) and the last layer as the output layer. All the layers in between (input and output) are called hidden layers.

In practice, each neuron from one layer, has access to all the neurons/inputs from the previous layer. During the training process, the weight of unrelated neurons/features becomes smaller.

Each neuron in the layer computes $\overrightarrow{w}.\overrightarrow{x} + b$ and passes the value to the next layer. Each neuron will learn a feature (such as affordability, quality, vertical line, etc.) by itself.

Each perceptron divides the space into two halves.

It is important to initialize $\overrightarrow{w}$ and b randomly, and not all equal to zero, otherwise, if all numbers are zero, there would be no training.

2.4.2 Logistic regression algo with Neural Network

We use a instead of f, which indicate the activation function. Each neuron in a layer does this calculations. To distinguish activations/parameters for each layer, we can use superscrip []. For example, $a_2^{[1]}$ denotes the second neuron in the first layer or the same for parameters $w_2^{[1]}$. The output of layer one $a^{[1]}$ is the input for the second layer $\overrightarrow{w}^{[2]}.\overrightarrow{a}^{[1]} + b^{[2]}$. Input layer is considered as layer 0. The input layers is considered as layer 0, so $\overrightarrow{x} = \overrightarrow{a}^{[0]}$. There are several option for the activation function g() (see 2.4.3), for example, for sigmoid function, we have:

$$a = f_{\overrightarrow{w},b}(\overrightarrow{x}) = g(z = \overrightarrow{w}.\overrightarrow{x} + b) = \frac{1}{1 + e^{-z}} \tag{2.9}$$

The calculation for the activation j of layer l is:

$$a_j^{[l]} = g(\overrightarrow{w}_j^{[l]}.\overrightarrow{a}^{[l-1]} + b_j^{[l]}) \tag{2.10}$$

$a^{[0]}$ refers to the first layer/input layer (X), and the last year is $\hat{y} = a^{[l]}$. Considering all the activation units in one layer:

$$a_{n^{[l]}\times 1}^{[l]} = g(z_{n^{[l]}\times 1}^{[l]} = \overrightarrow{w}_{n^{[l]}\times n^{[l-1]}}^{[l]}.\overrightarrow{a}_{n^{[l-1]}\times 1}^{[l-1]} + b_{n^{[l]}\times 1}^{[l]}) \tag{2.11}$$

The dimension of $dW^{[l]}$ $db^{[l]}$ is the same as $W^{[l]}$ and $b^{[l]}$.

The dimension of vectorized items are (m is the number of samples/training set, and b is the broadcast of the same b in the previous equation):

$$A_{n^{[l]}\times m}^{[l]} = g(Z_{n^{[l]}\times m}^{[l]} = \overrightarrow{W}_{n^{[l]}\times n^{[l-1]}}^{[l]}.\overrightarrow{A}_{n^{[l-1]}\times m}^{[l-1]} + b_{n^{[l]}\times m}^{[l]}) \tag{2.12}$$

Why we need a deep neural network instead of a shallow one? There are functions that you can compute with a small L-layer deep neural network that shallower networks require exponentially more hidden units to compute.

Based on these notations, the forward and backward propagation look like the following plot:

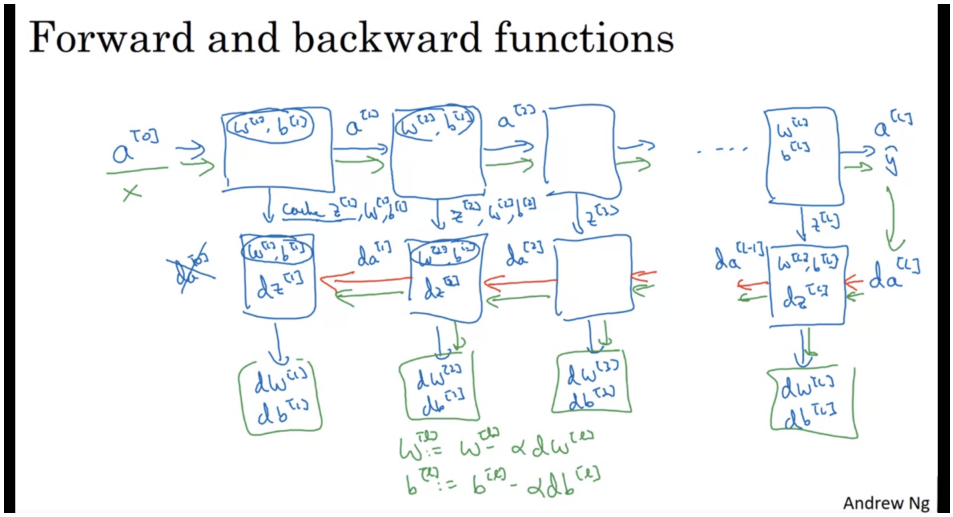


Figure 2.4: Forward and Backward functions

The backward equations look like this:

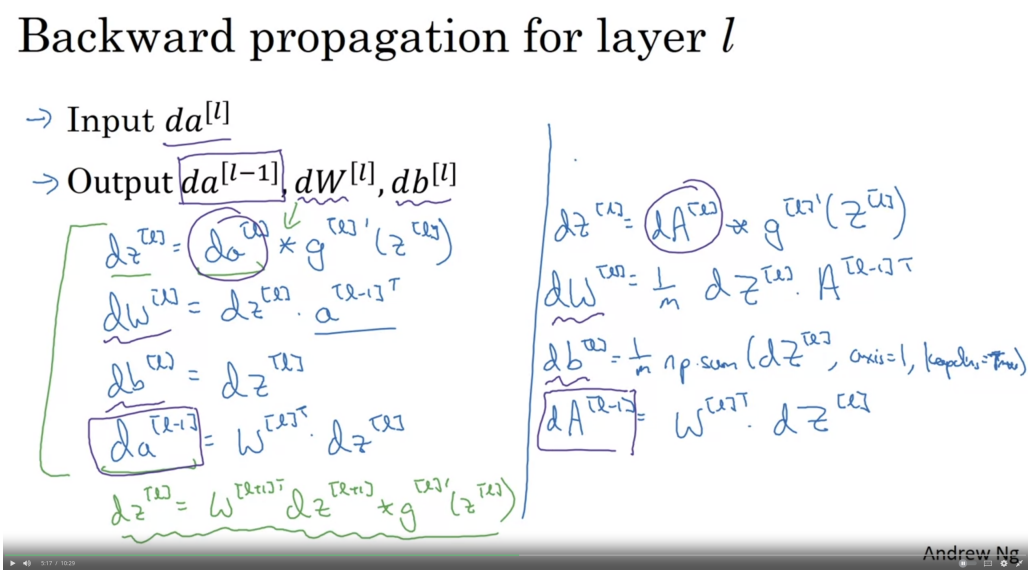


Figure 2.5: Forward and Backward functions

For inference (prediction), we only need a forward propagation.

2.4.3 Activation functions

Sigmoid function, for classification:

$$g(z) = \frac{1}{1 + e^{-z}} \quad \text{where} \quad 0 < g(z) < 1 \tag{2.13}$$

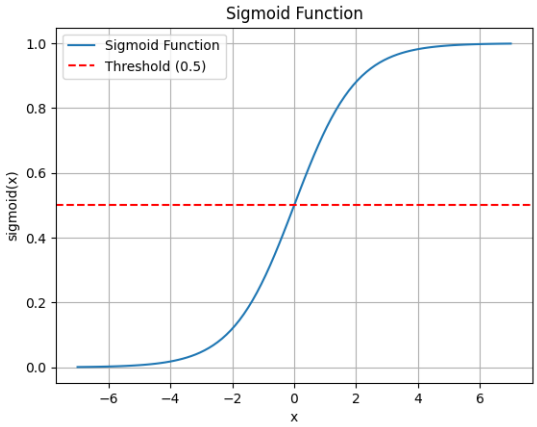


Figure 2.6: Sigmoid

Since the slope is close to zero at both ends of Sigmoid function, the gradient descent becomes very slow, but with ReLU, we don't have this issue.

Rectified Linear Unit (ReLU) function results in any non-negative value:

$$g(z) = max(0, z) = \begin{cases} z & z >= 0 \\ 0 & z < 0 \end{cases} \tag{2.14}$$

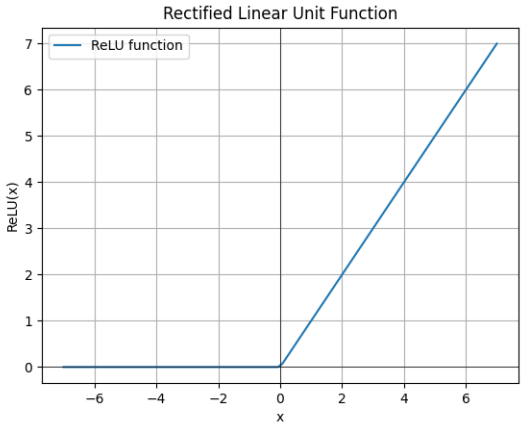


Figure 2.7: ReLU

Leaky Rectified Linear Unit (ReLU) function results in any non-negative value: Since the derivative is zero when z is negative, we may use a small number instead of zero, which works better in practice, but rarely being used.

$$g(z) = max(0, z) = \begin{cases} z & z >= 0 \\ smallnumber & z < 0 \end{cases} \tag{2.15}$$

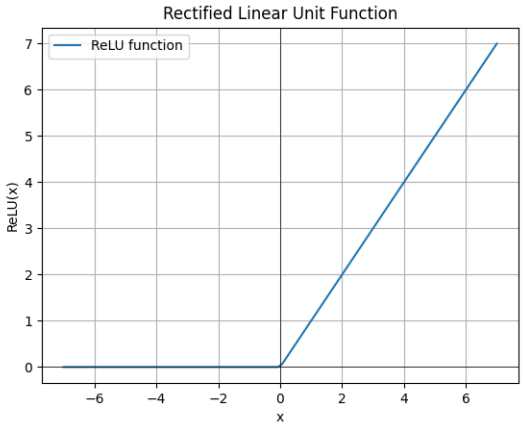


Figure 2.8: ReLU

Linear function, or in another words, not using any activation function:

$$g(z) = z \tag{2.16}$$

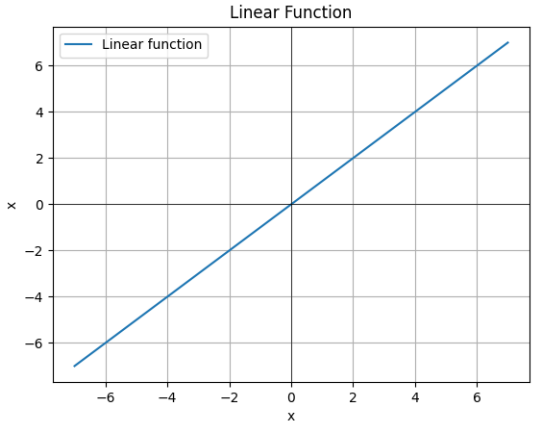


Figure 2.9: Linear function

Multiclass classification refers to classification problems where you can have more than just two possible output labels. The last layer has m activation units, corresponding to the m required classes, where each output is associated with a category/class.

Softmax function is a generalization of the logistic function. The last layer generates a vector of size m by a linear function which is applied to a softmax function. The softmax function converts the last layer into a probability distribution as described below. After applying softmax, each output will be between 0 and 1 and the outputs will add to 1, so that they can be interpreted as probabilities. The larger inputs will correspond to larger output probabilities. It provides probabilities of the input being in each category.

The last layer, then, has m activation units, each of which calculated as:

$$a_i = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_j}} \quad \text{where} \quad z_i = \overrightarrow{w}_i . \overrightarrow{x} + b_j \tag{2.17}$$

Note that $\sum_{i=1}^{m} a_i = 1$

The name softmax is coming from the contrast function hard-max, which converts the last year, to a vector with only one true / 1 label, and the rests would be zero. But with the softmax, all elements have a value and the sum of all values are one. Note also that if number of softmax classes is two, the softmax reduces to logistic regression.

Tanh function, LeakyReLU function, and swish function are other choices.

Multi-label classification (different from multiclass classification): for example, in image processing we want to know if there is a bus/car/pedestrian. One way to have three model/NN for each label. The other way is to train one model/NN to simultaneously detect all three. For that, the last layer should have three units with a sigmoid function for each of those last year units.

How to select activation function? The selection of activation function for the final/output layer, depends on the type of target. For example for binary classification, we need sigmoid. For regression, we can use linear activation function. If the target is a positive number, such as the price of a house, ReLU is the natural choice.

For hidden layers though, ReLU is a better choice. The reason is that ReLU function is faster to computer. Next reason is that ReLU is only flat in one side of the function but sigmoid is flat on both sides, which makes gradient descend very slow.

Why do we need activation function? A neural network with no activation function is exactly the same as a linear regression method, and multiple layers is not helping to improve the neural net. To create non-linearity in the cost function.

2.4.4 Biases

What kind of problems are neural networks appropriate for solving?

Restriction Bias A neural network's restriction bias (which, if you recall, is the representation's ability to consider hypotheses) is basically non-existent if you use sigmoids, though certain models may require arbitrarily-complex structure. We can clearly represent Boolean functions with threshold-like units. Continuous functions with no "jumps" can actually be represented with a single hidden layer. We can think of a hidden layer as a way to stitch together "patches" of the function as they approach the output layer. Even arbitrary functions can be approximated with a neural network! They require two hidden layers, one stitching at seams and the other stitching patches.

This lack of restriction does mean that there's a significant danger of overfitting, but by carefully limiting things that add complexity (as before, this might be layers, nodes, or even the weights themselves), we can stay relatively generalized.

Preference Bias On the other hand, we can't yet answer the question of the pref- erence bias of a neural network (which, if you recall, describes which hypotheses from the restricted space are preferred). We discussed the algorithm for updating weights (gradient descent), but have yet to discuss how the weights should be initialized in the first place.

Common practice is choosing small, random values for our initial weights. The randomness allows for variability so that the algorithm doesn't get stuck at the same local minima each time; the smallness allows for relative adjustments to be impactful and reduces complexity. Given this knowledge, we can say that neural networks—when all other things are equal—prefer simpler explanations to complex ones. This idea is an embodiment of Occam's Razor: Entities should not be multiplied unnecessarily. More colloquially, it's often expressed as the idea that the simpler explanation is likelier to be true.

2.4.5 Convolutional Neural Network (CNN)

In this method, each neuron only looks at part of the previous layer's outputs, instead of a regular layer, which each neuron has weights for all previous layer neurons. The main advantage of this method is that 1) it is faster, 2) Need less training data and less prone to overfitting.

2.4.6 Tuning hyperparameters

- learning rate (most important parameter #1) α
- momentum term (#2)β
- Adam optimization $\beta_1, \beta_2, \epsilon$ (#4)
- number of layers (#3)
- number of units in each layer (#2)
- learning rate decay (#3)

- mini-batch size (#2)

To investigate the hyperparameters, use random points rather than a grid. You may need to test these with many hyperparamters simultaneously. The other technique is to go from a coarse to fine regions.

Using an appropriate scale to pick hyperparamters matters, for example, for number of layers/units, we can use an equally spaced grid or linear scale, but for the learning rate we need to use logarithmic pattern ($\alpha = 10^{-4 \times np.random.rand()}$).

For exponentially weighted average, we can use $1 - \beta$, sample uniformly between 0.1 to 0.001 $\beta = 1 - 10^{-3 \times np.random.rand()}$.

Some of the hyperparameters have independent effects on the outcome. This is called orthogonalization. Experts know what knob to turn, to get what they want in a model. Here is the chain of assumptions in ML:

- fit training set well on cost function (human-level performance). If not working well:
  - use a bigger network
  - use more efficient optimization, such as Adam
- fit dev set well on cost function. if not working well:
  - use regularization
  - use bigger training set
- fit test set well on cost function. if not working well:
  - use bigger dev set
- performs well in real world
  - change dev set
  - change cost function

Early stopping method might not be good option.

2.4.7 Error analysis

Deep learning models are quite robust to random errors in the training set, but very sensitive to the systematic errors.

2.5 SL: Sequence models-Recurrent Neural Network (RNN)

RNN is a neural network that deals with the different amount of inputs. Similar to the regular NN, RNN has bias, weights, and activation functions. The only difference is that RNN has feedback loops, which allows prior values influence the prediction. Examples of sequence data with different amount of input: stock market predictions, speech recognition, music generation, sentiment classification, DNA sequence analysis, machine translation, video activity recognition, and name entity recognition (finding the names in a sentence).

The main idea is that we have the same NN, and for each each input we repeat/unroll the same NN and add the output from the previous input/NN to the next one. Regardless of how many times we unroll the neural network, weights and biases are shared across every input. One mathematical issue with the RNN is the ***vanishing/exploding gradient problem***. As the number of input increases, if the weight is greater than one, we keep multiplying, until the number explodes, and this results is very large steps in the gradient descent. The same way, if weights are less than one, after some multiplication, the number vanishes, and in the gradient descent, this results in very small steps.

The downside of RNN is that it only uses the info before the current layer (This is for unidirectional method, in the bidirectional, we move in both directions). Here is the forward propagation:

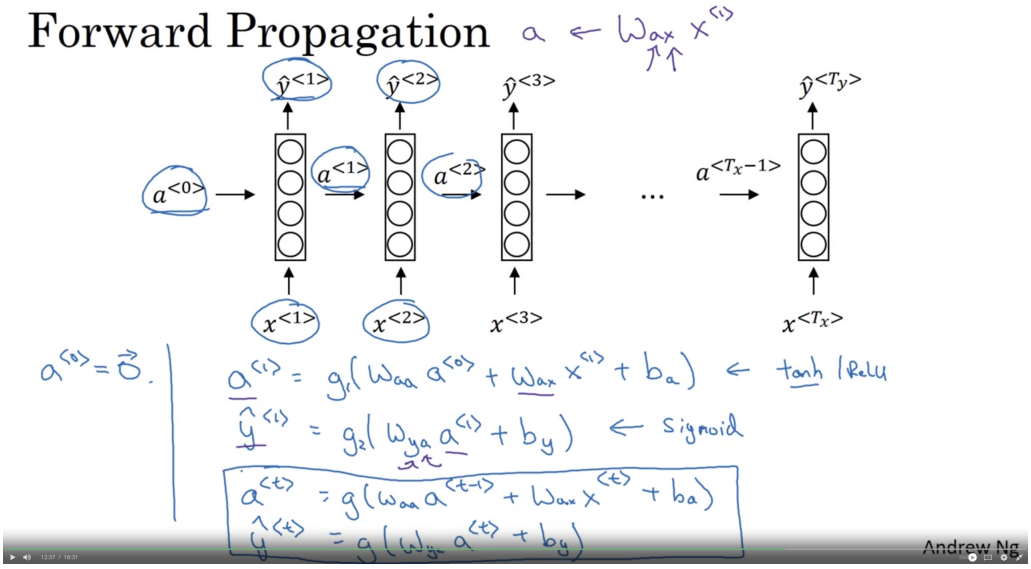


Figure 2.10: Forward propagation chart for RNN

Here is the backpropagation model:

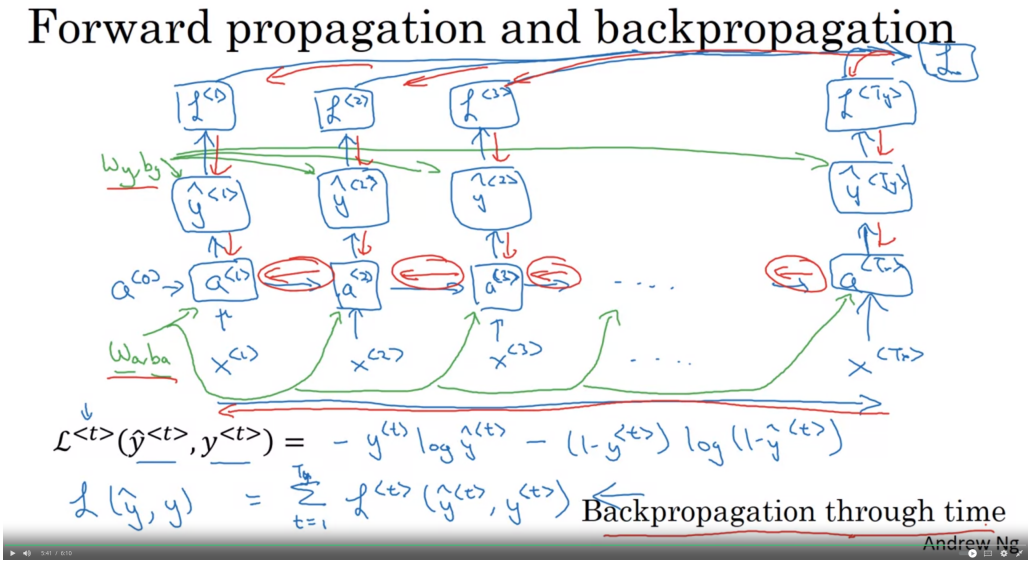


Figure 2.11: Back propagation chart for RNN

There are five different types of RNN model, as depicted in the following figure:

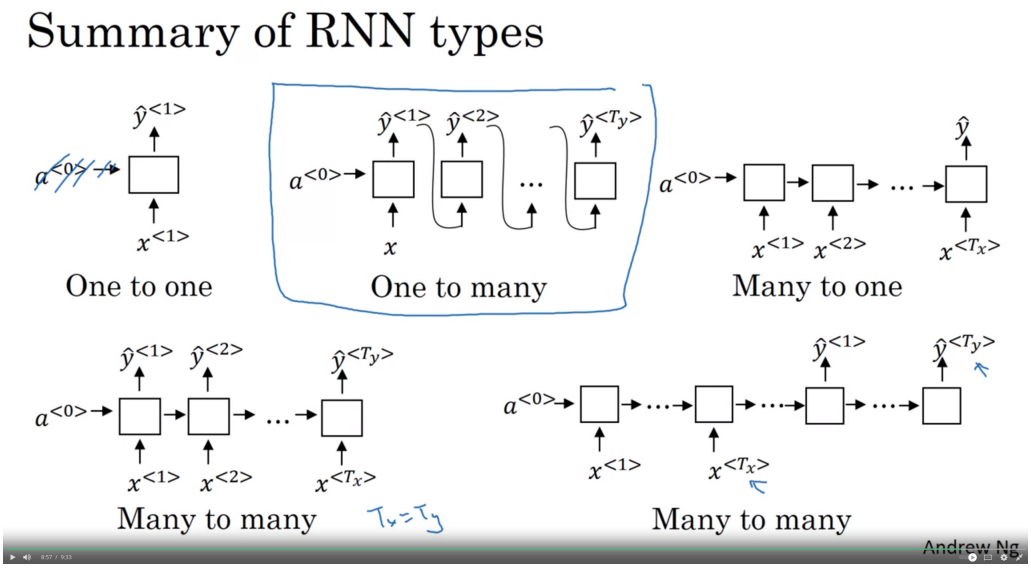


Figure 2.12: Various RNN types

One main application of the DRNN is in NLP models. There is a dictionary/vector with all the words, and each word of the sentence is translated to a one-hot vector of the size of the dictionary, with all zeros, except for the index of the word which is one. Then, each sentence is a matrix where the number of columns is equal to the number of words in the sentence. For the words that are not in the dictionary, we can add one token to the dictionary for unknown words. Since every sentence has a different length, we cannot just train a regular NN.

From chatGPT: which ML model is used for LLM: " Transformer-based architectures, including those used in large language models like GPT, are not based on Recurrent Neural Networks (RNNs). Instead, they rely on a different architecture known as the transformer.

RNNs are sequential models that process input data one element at a time while maintaining an internal state that represents information from previous elements. While RNNs have been widely used in natural language processing tasks, they have limitations in capturing long-range dependencies due to the sequential nature of their computation and the vanishing gradient

problem.

Transformers, on the other hand, are designed to handle sequential data more efficiently by employing self-attention mechanisms. This allows them to capture relationships between different elements of the input sequence simultaneously, making them better suited for tasks involving long-range dependencies. Transformers have gained popularity in recent years due to their superior performance on various natural language processing tasks, including language modeling, machine translation, and text generation. ”

Vanishing and exploding gradients are major challenges of RNN. One way to avoid that is Gated Recurrent Unit (GRU).

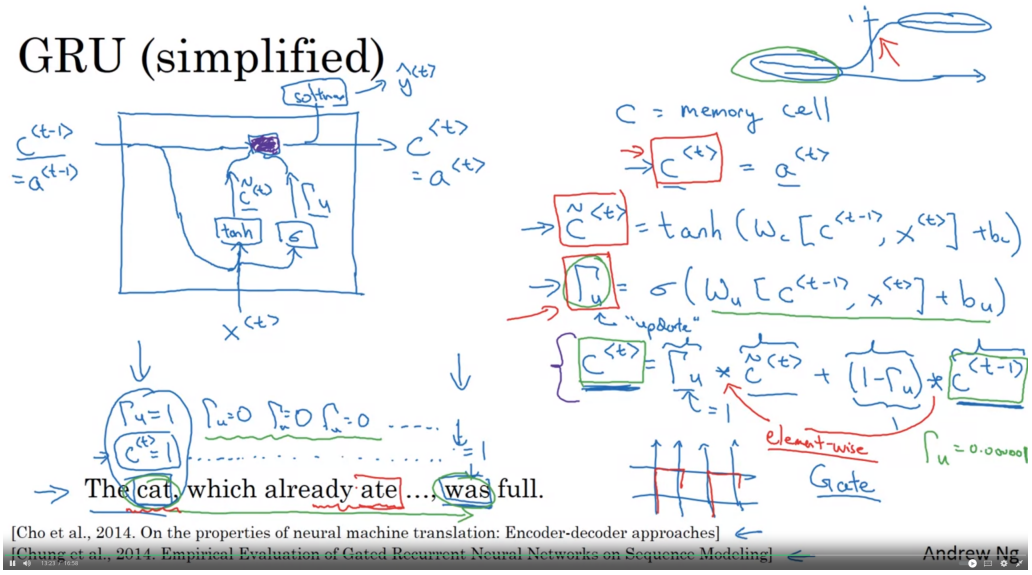


Figure 2.13: Various RNN types

Long Short-Term Memory (LSTM) is similar to GRU,

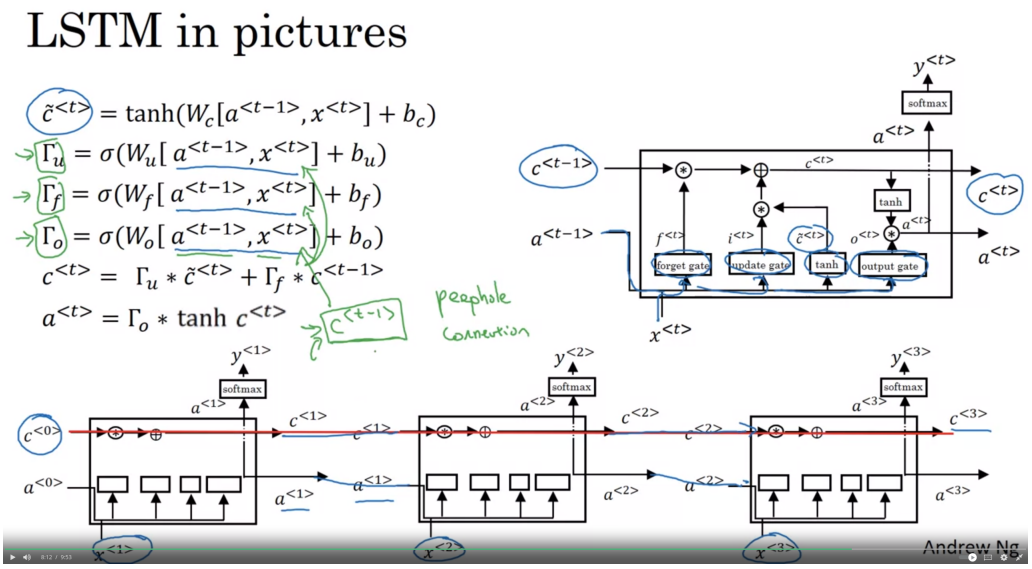


Figure 2.14: LSTM

Bidirectional RNN lets you at the point in time to take information from both earlier and later in the sequence.

Deep RNNs have hidden layers:

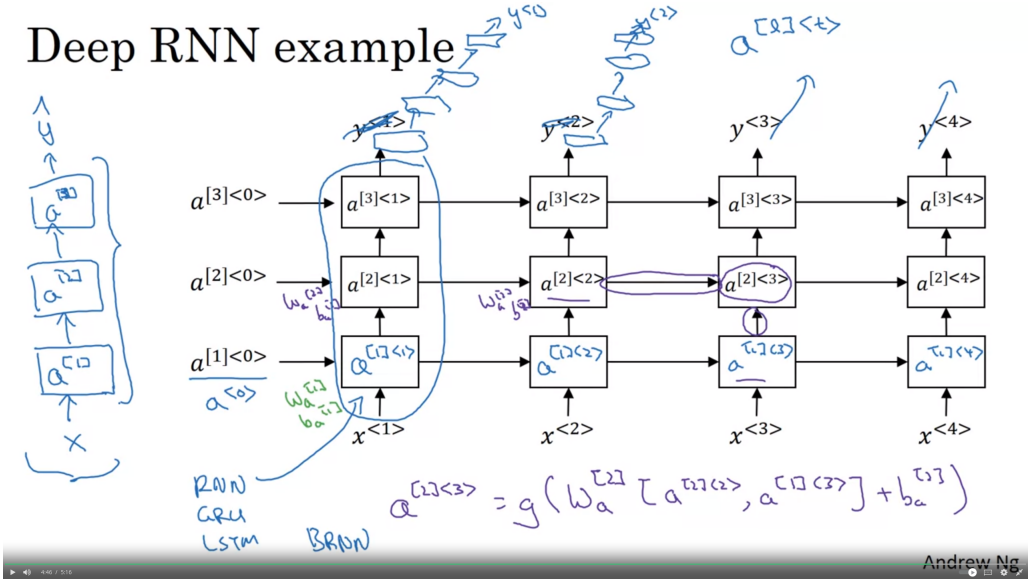


Figure 2.15: LSTM

2.6 SL: Instance based learning

2.7 SL: Ensemble B&B

Ensemble Learning is in general the process of combining some simple rules into a more complex rule that can generalize well.

It works by dividing the data into smaller subsets, learn over each individual subset to come up with a rule, then combine all these rules in a single more complex rule. If we looked at the whole data, it would be hard to come up with simple rules.

2.8 SL: Kernel methods and SVMs

2.9 SL: Comp Learning Theory

2.10 SL: VC dimensions

2.11 SL: Bayesian Learning

2.12 SL: Bayesian inference

Chapter 3

Unsupervised learning (UL)

Unsupervised algorithms don't make use of a target; instead, their purpose is to learn some property of the data, to represent the structure of the features in a certain way. In the context of feature engineering for prediction, you could think of an unsupervised algorithm as a "feature discovery" technique.

3.1 Clustering

Clustering simply means the assigning of data points to groups based upon how similar the points are to each other. When used for feature engineering, we could attempt to discover groups of customers representing a market segment, for instance, or geographic areas that share similar weather patterns. The other examples of using clustering is grouping similar news, DNA analysis, astronomical data analysis.

The motivating idea for adding cluster labels is that the clusters will break up complicated relationships across features into simpler chunks. Our model can then just learn the simpler chunks one-by-one instead having to learn the complicated whole all at once. It's a "divide and conquer" strategy.

There are multiple methods for clustering. They differ primarily in how they measure *similarity* or *proximity* and in what kinds of features they work with. Clustering algorithms include: Distribution-based methods (expectation-maximization algorithm), Centroid-based methods (K-means), Connectivity-based methods (hierarchical algorithms), Density model (DBSCAN), and subspace clustering.

3.1.1 K-means clustering

It measures similarity using ordinary straight-line distance (Euclidean distance). It creates clusters by placing a number of points, called centroids, inside the feature-space. Each point in the dataset is assigned to the cluster of whichever centroid it's closest to. The "k" in "k-means" is how many centroids it creates.

Clustering has two main steps, selecting or moving centriods, and then assigning nodes to centriods. To do the clustering, we randomly select k points/centroids in the space, not necessarily a real point from the data set, but just in the space3. Then, it calculates the distance between each node and the centroids, and assigns each node to closest centriod. For the next iteration, we need to adjust the centriod. First, we find the average of each cluster points, and use that one as the new centriod. We repeat the process, until the point is not moving. In the first step, we keep the centriods fixed and just change the point assignments. In the next step, centriods change. This method is guaranteed to converge.

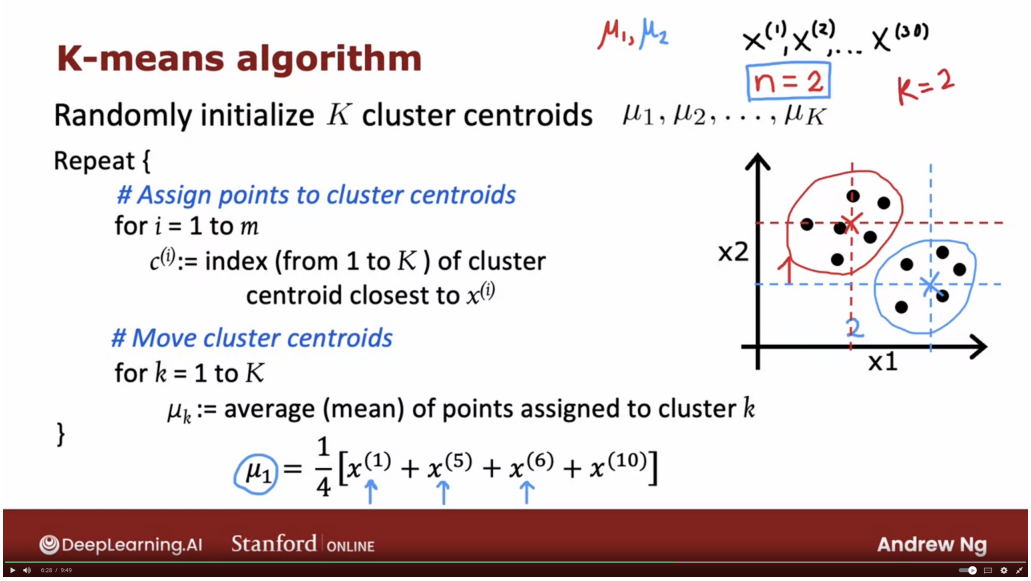


Figure 3.1: K-mean clustering

The only exception is that if there is no point assigned to a centriod, which means calculating the average is not defined. In that case, we either eliminate

the cluster point, or randomly assign a new point to the cluster. Sometimes, the points are not really separated/clustered. In that case, theoretically, it is more difficult to decide how many points are needed.

The K-mean algorithm is indeed an optimization problem, based on the distortion cost function.

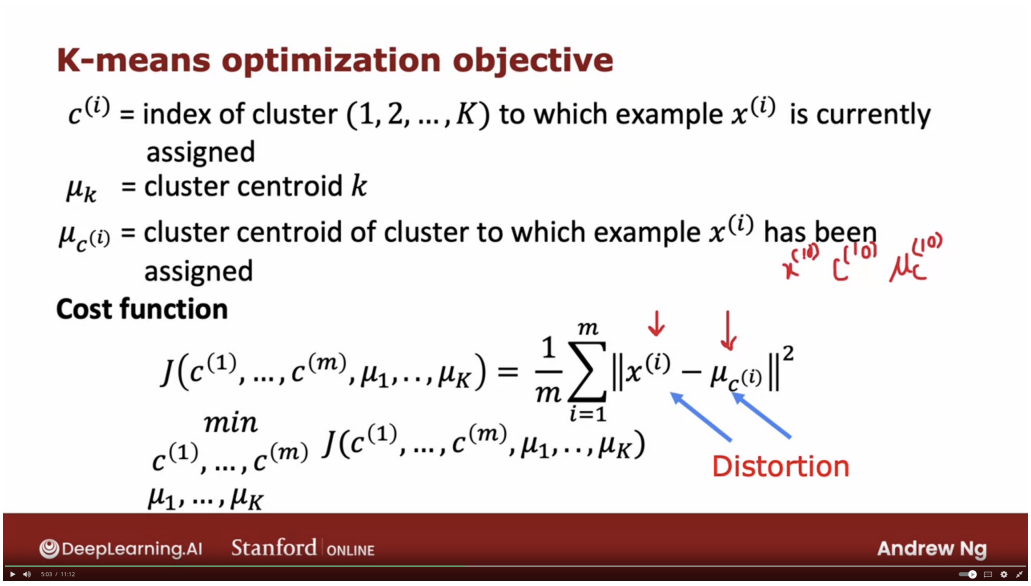


Figure 3.2: K-mean clustering distortion optimizer

The very first step of the K-means algorithm is the initial guess for the number of clusters and the location of centriods. K should definitely be less than the number of points. To find out the best K for a data set, we can test various K numbers and plot K vs variance, this is called an *elbow plot*. The best K is the one that the slope of variance reduces. If K is equal to the number of items in the dataset, then, the variance would be zero. The other factor is that how you want to use the clusters, the larger the K is, may cause a higher cost.

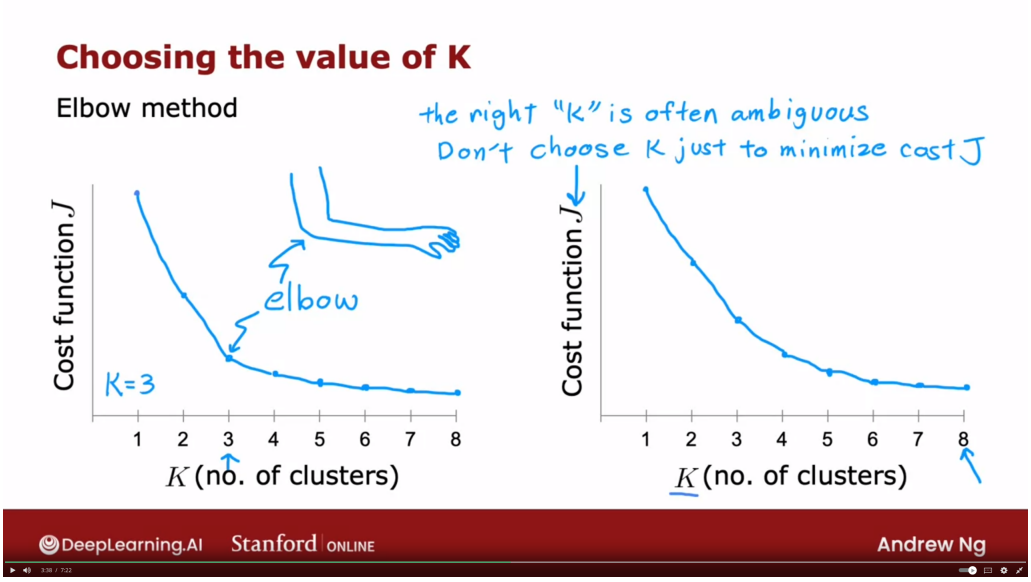


Figure 3.3: K-mean clustering, K selection

For the location of centriods, we can randomly choose from nodes. The downside of this method is that we may stuck in the local minimum. To avoid that, we can run this process multiple times and select the one with the lowest cost.

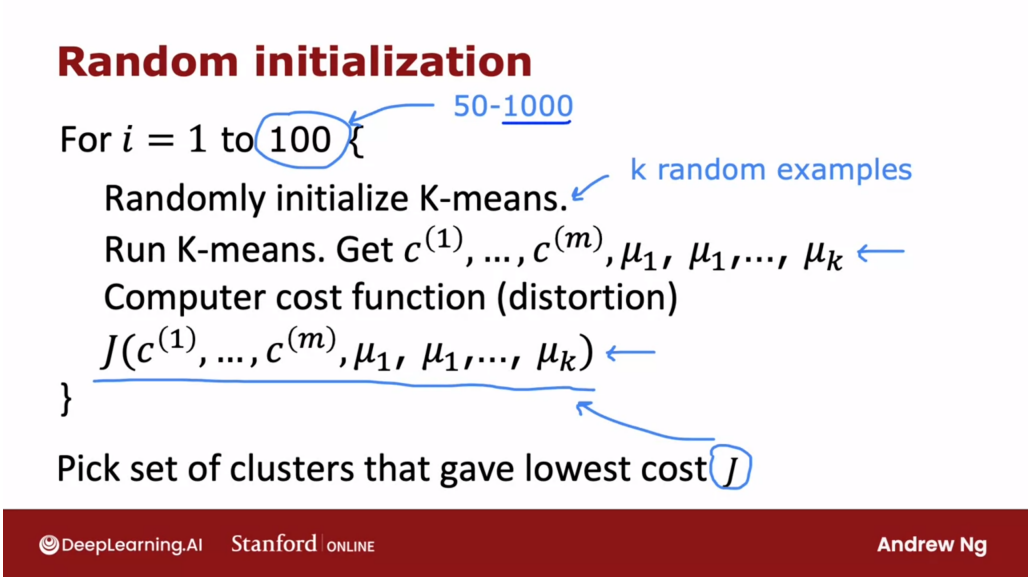


Figure 3.4: K-mean clustering initialization

The k-means algorithm is sensitive to scale. This means we need to be thoughtful about how and whether we rescale our features since we might get very different results depending on our choices. As a rule of thumb, if the features are already directly comparable (like a test result at different times), then you would *not* want to rescale. On the other hand, features that aren't on comparable scales (like height and weight) will usually benefit from rescaling. Sometimes, the choice won't be clear though. In that case, you should try to use common sense, remembering that features with larger values will be weighted more heavily.

What you should take away from this is that the decision of whether and how to rescale features is rarely automatic – it will usually depend on some domain knowledge about your data and what you're trying to predict. Comparing different rescaling schemes through cross-validation can also be helpful.

3.1.2 Principal Component Analysis (PCA)

Just like clustering is a partitioning of the dataset based on proximity, you could think of PCA as a partitioning of the variation in the data. PCA is a great tool to help you discover important relationships in the data and can

also be used to create more informative features. PCA is typically applied to standardized data. With standardized data "variation" means "correlation". With unstandardized data "variation" means "covariance". All data in this course will be standardized before applying PCA. The main idea of PCA: instead of describing the data with the original features, we describe it with its axes of variation. The axes of variation become the new features. The new features PCA constructs are actually just linear combinations (weighted sums) of the original features. These new features are called the principal components of the data. The weights themselves are called loadings.

PCA also tells us the amount of variation in each component. We can see from the figures that there is more variation in the data along the Size component than along the Shape component. PCA makes this precise through each component's percent of explained variance. It's important to remember, however, that the amount of variance in a component doesn't necessarily correspond to how good it is as a predictor: it depends on what you're trying to predict.

3.2 Anomaly detection

The idea is to build and train a (unsupervised) model based on some samples and their features, then the task for this model is to review any new sample and find out if there is any anomaly with the features. The method for this task is called density estimation. In this method, we find regions with different levels of probability/anomaly, in which, if the new sample features locate in the high probability area, there is no anomaly, but as we move away from it, anomaly probability increases. This method has several applications including fraud detection, manufacturing, monitoring computers in data centers, etc.

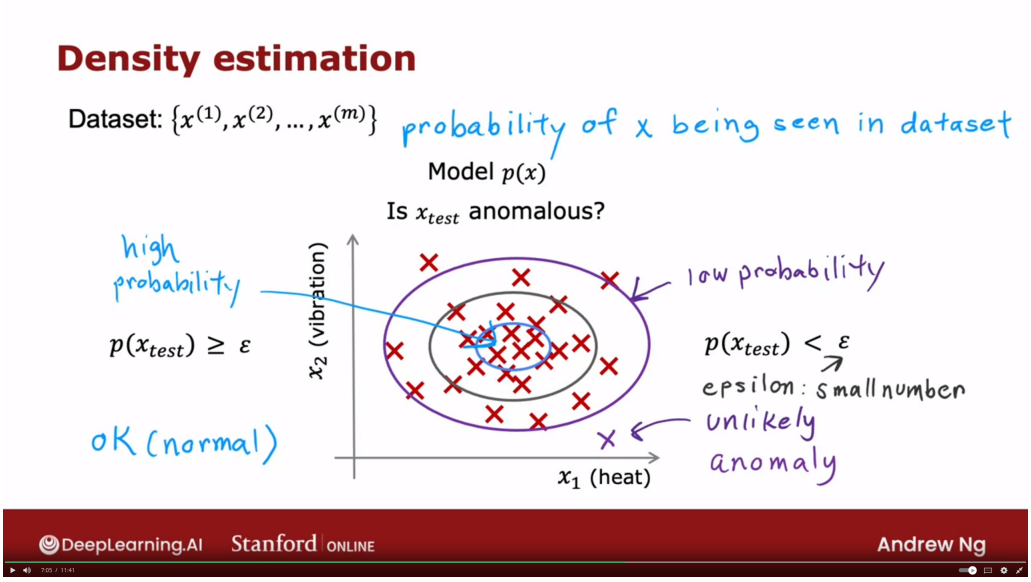


Figure 3.5: Density estimation

To apply anomaly detection, we use Gaussian (normal/bell shape) distribution. The next picture shows the distribution equation:

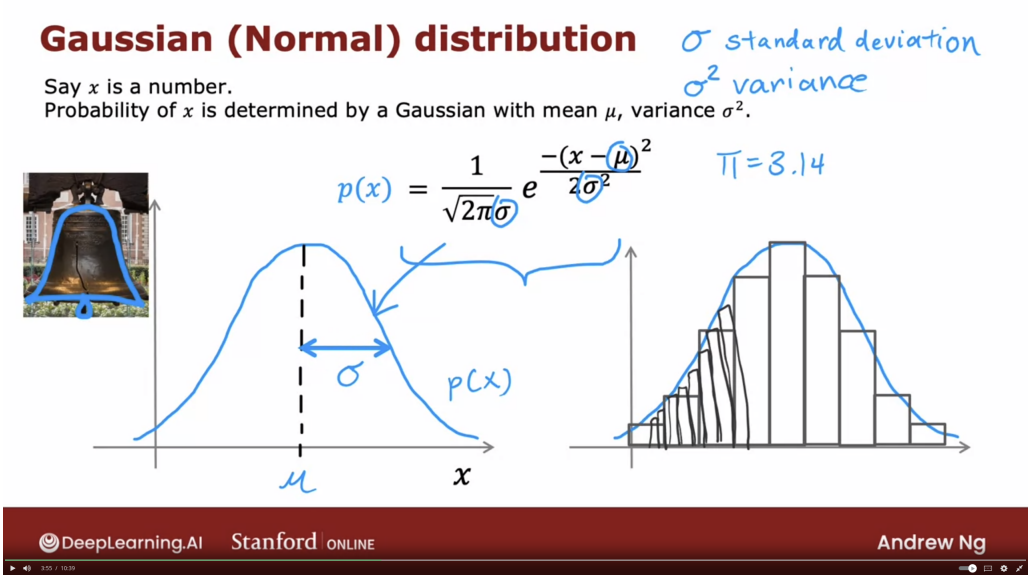


Figure 3.6: Gaussian distribution

To calculate the Gaussian distribution for a set of data:

$$\mu = \frac{1}{m}\Sigma_{i=1}^{m} x^{(i)} \sigma^2 = \frac{1}{m}\Sigma_{i=1}^{m}\left(x^{(i)} - \mu\right)^2 \tag{3.1}$$

but notice that each sample might have multiple features. To calculate the probability of one sample, then we need to multiply the probability of each each feature. Multiplication is only valid if the features are statically independent, which is usually not the case, but for this purpose, it is fine to use multiplication, even though features are dependent. Here is how it works:

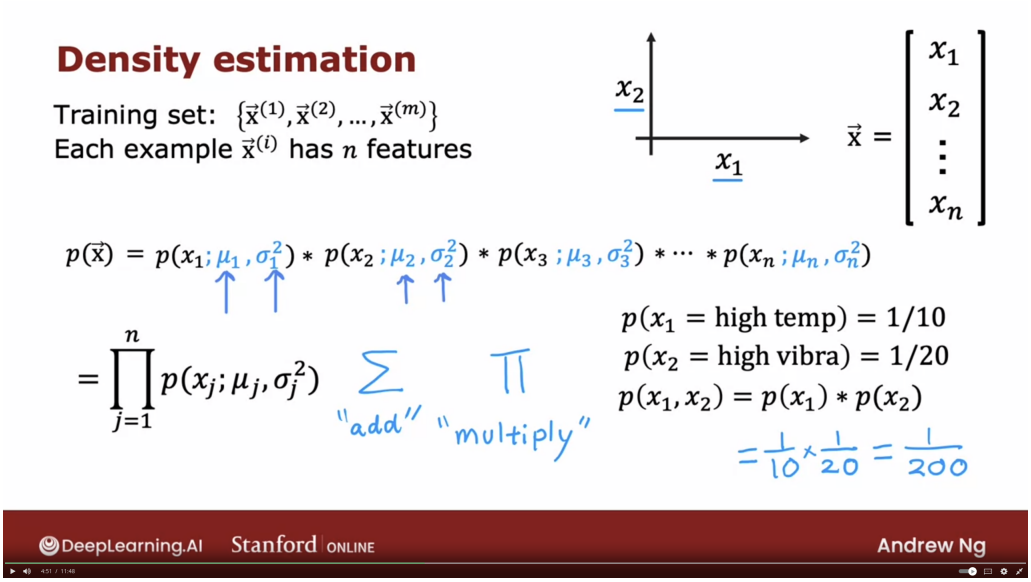


Figure 3.7: Gaussian distribution

In summary, here are the steps for anomaly detection:

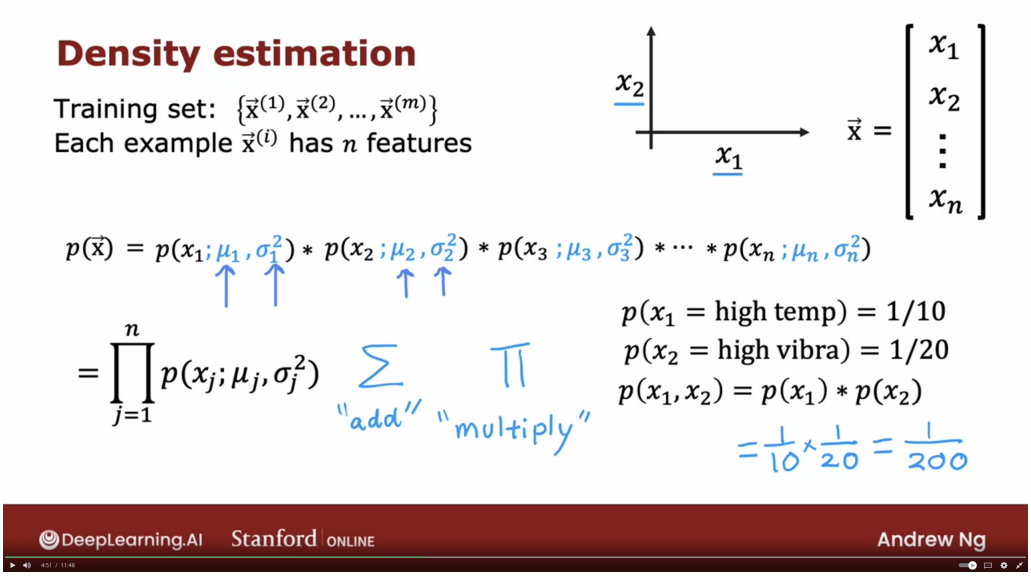


Figure 3.8: Gaussian distribution

To train the anomaly detection system, as always, we need to partition the data set. Let's say we have a data set with 10000 normal cases and 20, anomalies. We can partition the data as follows, then we use the training set to find $\mu \& \sigma^2$, and we use the cross validation algorithm to find and tune ϵ and even add or subtract some features. The test set is to report on the accuracy of the model, but if the set is small, we can skip the test set, and put all the anomalies in the cross validation set.

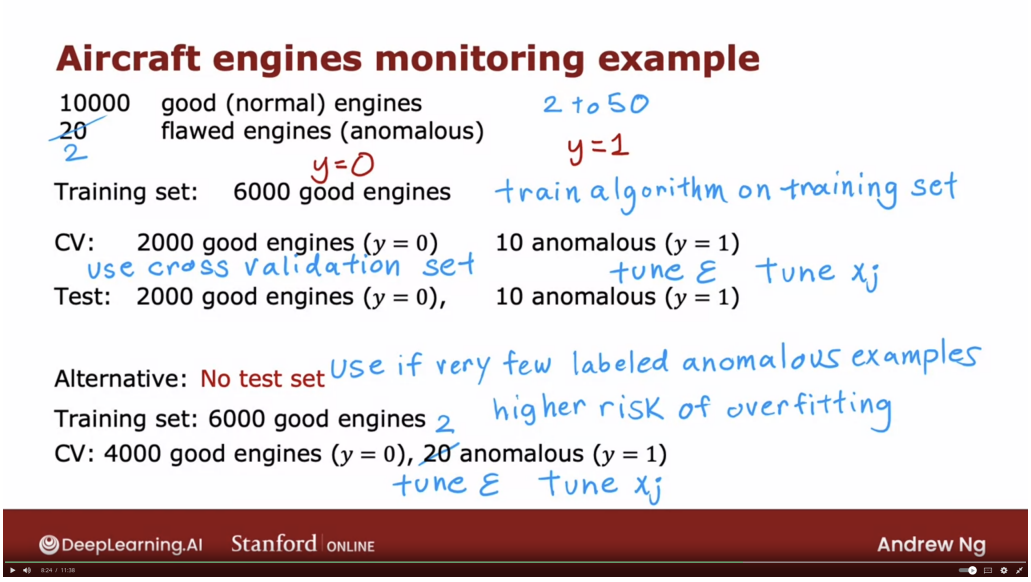


Figure 3.9: Gaussian distribution

To predict the accuracy of the model, we use the cross/test sets and see how many anomalies are flagged, and how many normal engines are flagged as anomalies.

This is still an unsupervised approach, because, technically samples don't have a label, but we just have a few anomalies labels. The question is why if we have anomalies label, not to use a supervised approach then. The way anomaly detection looks at the data set versus the way supervised learning looks at the data set are quite different. If you think there are many different types of anomalies or many different types of positive examples, then anomaly detection might be more appropriate. If there may be a brand new way anomaly then your 20 say positive examples may not cover all of the anomaly types. That makes it hard for any algorithm to learn from the small set of positive examples. And future anomalies may look nothing like any of the anomalous examples we've seen so far. If you believe this to be true for your problem, then I would gravitate to using an anomaly detection algorithm rather than supervised learning. In contrast, if there are enough positive examples that cover all types of anomalies, then a supervised algo is more appropriate. Anomaly detection tries to find brand new types that might not have been seen before.

In anomaly detection, choosing the features is one main question (this is

not the case with supervised learning, as less important features will phase out in the learning process). Features should be Gaussian, and if they are not, we should change them to be look like more to Gaussian distribution:

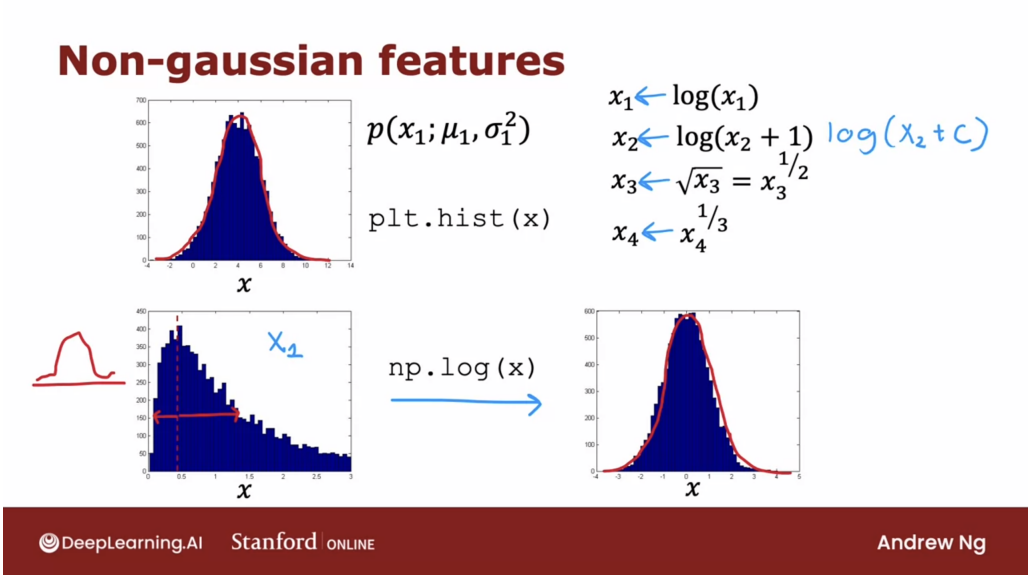


Figure 3.10: Gaussian distribution

Chapter 4

Reinforced Learning (RL)

4.1 Markov Decision Process

Chapter 5

Large Language Models (LLM) and Transformers

5.1 Introduction

Generative AI: A type of AI that can produce various types of content, including text, imagery, audio, and synthetic data based on what it has learned from existing content. Gen AI is a subset of Deep Learning, which uses Artificial Neural Networks, can process both labeled and unlabeled data, using supervised, unsupervised, and semi-supervised methods.

DL/ML can be divided into two types: generative and discriminative/predictive. **Discriminative model** is used to classify or predict labels for data points. They are typically trained on a dataset of labeled data points, and they learn the relationship between the features of the data points and the labels. After training, discriminative models can be used to predict/cluster/classify the label for new data points. **A generative model** generates new data instances based on a learned probability distribution of existing data.

In short, the output of a discriminative model is a number, probability, class, and discrete. But the output of generative AI is natural language, audio, or image.

The power of Generative AIs is coming from transformers, announced for the first time in a paper called "Attention is all you need", which proposed a neural network architecture that replaces traditional recurrent neural networks (RNN and CNN). The Transformer architecture consists of an encoder and a decoder, each of which is composed of several layers. Each layer consists

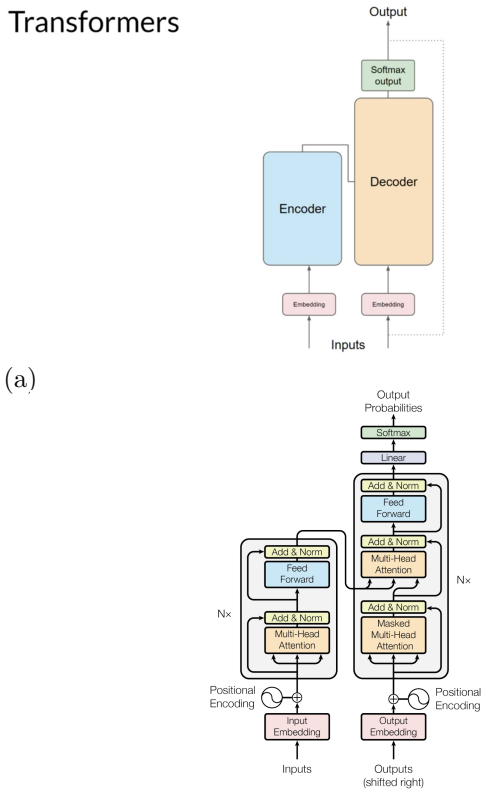


Figure 5.1: Transformer structure: (a) Encoder/decoder section, (b) Full picture

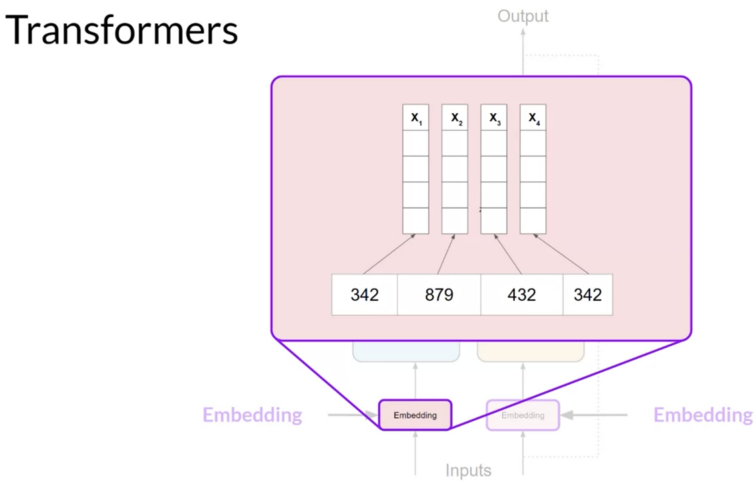
of two sub-layers: a multi-head self-attention mechanism and a feed-forward neural network. The multi-head self-attention mechanism allows the model to attend to different parts of the input sequence, while the feed-forward network applies a point-wise fully connected layer to each position separately and identically. The Transformer model also uses residual connections and layer normalization to facilitate training and prevent overfitting. In addition, the authors introduce a positional encoding scheme that encodes the position of each token in the input sequence, enabling the model to capture the order of the sequence without the need for recurrent or convolutional operations. A transformer is a special type of neural network.

Step 1: ML models are just big statistical calculators and they work with numbers, not words. So, before passing texts into the model to process, you

must first tokenize the words. This converts the words into numbers with each number representing a position in a dictionary of all the possible words that the model can work with. There are multiple tokenization methods. For example, token IDs matching the entire word, or using token IDs to represent parts of words. We need to use the same tokenizer for encoding and decoding.

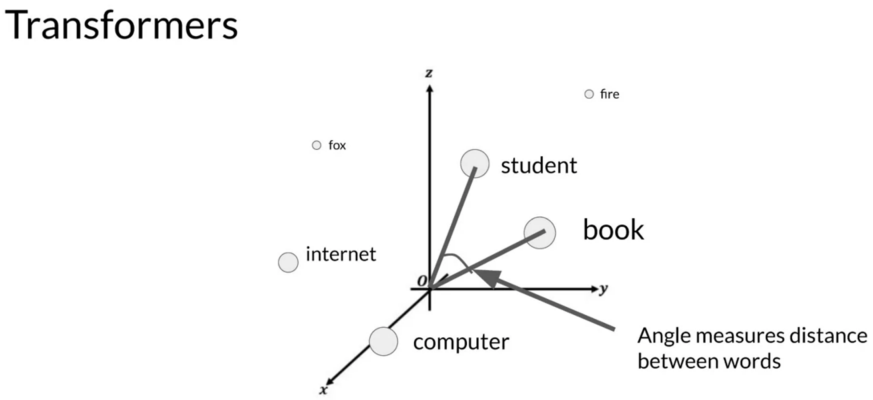
Step 2: Once the input is tokenized, you can pass it to the embedding layer. This is a trainable vector embedding in space. Each token ID is represented as a vector and is mapped to a multi dimensional vector. These vectors learn to encode the meaning and the context of the individual tokens in the input sequence. Words with similar meanings land in a space closer to each other.

Figure 5.2: Embedded spaces in transformers



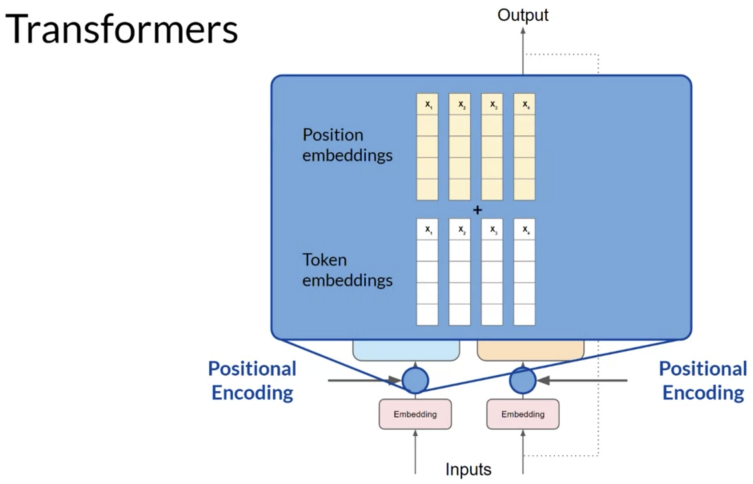
The embedded space allows you to find the relation ship between spaces. For example, assume that the space is only three dimensional (the actual paper uses a 512-dimensional space.). Then, you find related words in the space.

Figure 5.3: Example of embedded spaces



Step 3: Positional encoding. The model processes each of the input tokens in parallel. So, bu adding the positional encoding, you preserve the information about the word order and don't lose the relevance of the position of the word in the sentence. Once you summed the input tokens and the positional encodings, you pass the resulting vectors to the self-attention layer.

Figure 5.4: Positional encoding

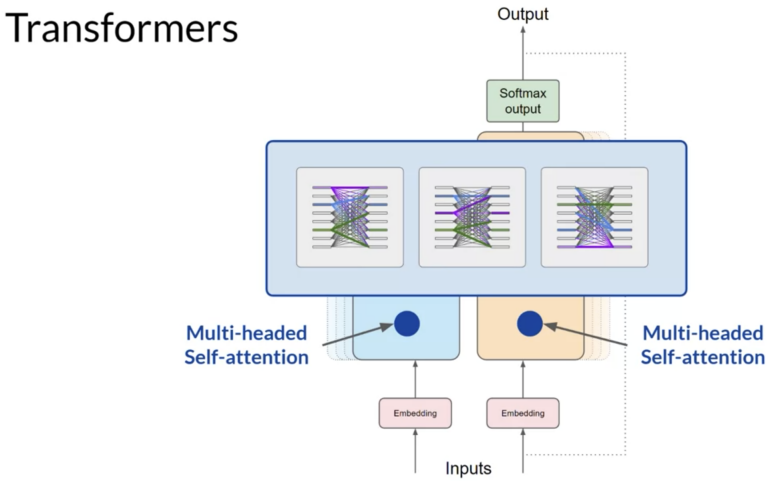


Step 4: Self-attention layer. The model analyzes the relationships be-

tween the tokens in your input sequence. This allows the model to attend to different parts of the input sequence to better capture the contextual dependencies between the words. The self-attention weights that are learned during training and stored in these layers reflect the importance of each word in that input sequence to all other words in the sequence.

Step 5: Multi-headed self-attention. Multiple sets of self-attention weights or heads are learned in parallel independently of each other,. The number of attention heads included in the attention layer varies from model to model, but number in the range 12 100 are common. Each self-attention head will learn a different aspect of language. For example, one head may see the relationship between the people entities in our sentence. Whilst another head may focus on the activity of the sentence. Another head, may search for other properties. The weights of each head are randomly initialized, so each, randomly, learn different aspect of the language.

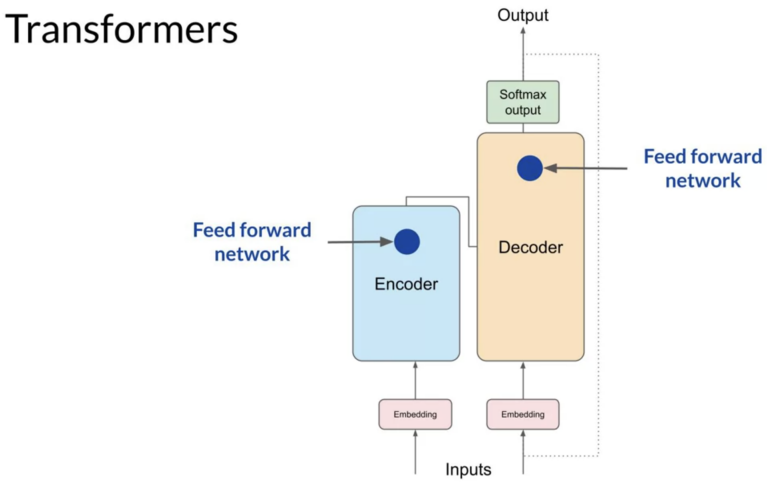
Figure 5.5: Multi-headed self-attention



Step 6: The output is processed through a fully connected feed forward network. The output of this layer is a vector of logics proportional to the probability score for each and every token in the tokenizer dictionary. You can pass these logics to the final softmax layer, where they are normalized into a probability score for each word. This output includes a probability for every single word in the vocabulary, so there is likely to be thousands of scores here. One single token will have a score higher than the rest. This is

the most likely predicted token. There are a number of methods that you can use to vary the final selection from this vector of probabilities.

Figure 5.6: Feed forward network



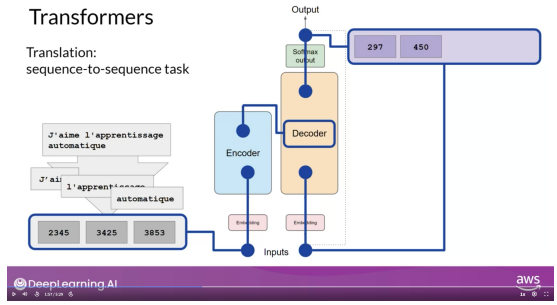
The process is depicted in the following picture, the process is iterative, until the end of the process, then, the final sequence of tokens are detokenized into words and you get the output

5.2 Transformer types

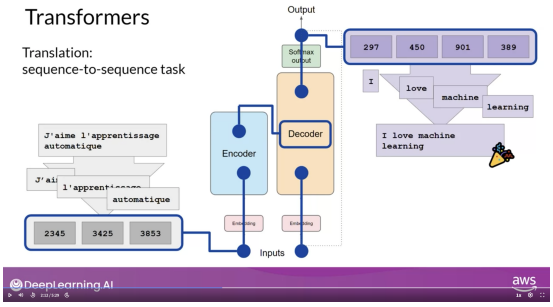
There are three different transformer models (see the picture below).

Some models only uses the Encoder section of the transformer (Encoder Only Models, AKA autoencoding models) by adding some extra layers to the output. These models are trained used Masked Language Modeling (MLM), which randomly masks some tokens from the sentence, and tries to guess the masked token. The model is bidirectional (from the beginning to end and end to the beginning of the sentence). These models are good for Sentiment analysis, Name entity recognition, and word classification. Famous autoencoding models are BERT and ROBERTA.

Decoder only models (autoregressive models) are trained using Causal Language Modeling (CLM). The training objective is to predict the next token based on the previous sequences of tokens (some researchers call this full language modeling). These models mask the input sequence and can only



(a)



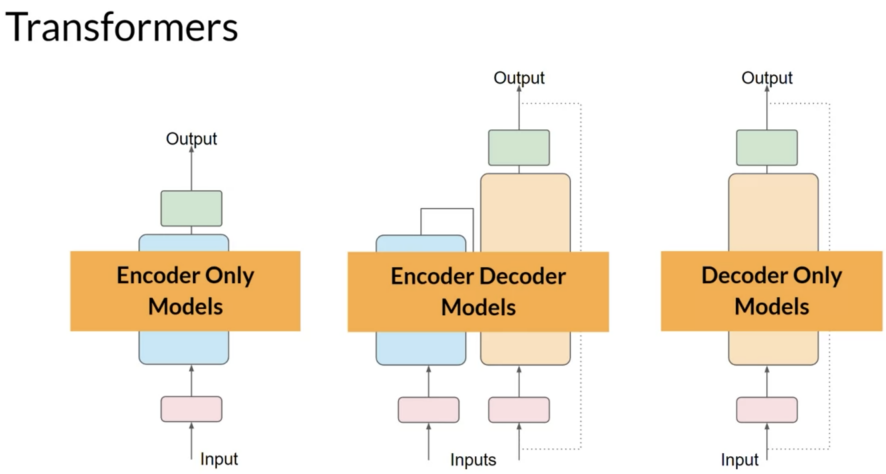
(b)

Figure 5.7: Full transformer process: (a) Iterations of transformer model, (b) Detokenization of the final sequence

see the input tokens leading up to the token in question. The model has no knowledge of the end of the sentence and the context is unidirectional. These models are good for text generation, and other emergent behavior (depends on model size). Examples of these models are GPT, BLOOM, and LLaMa.

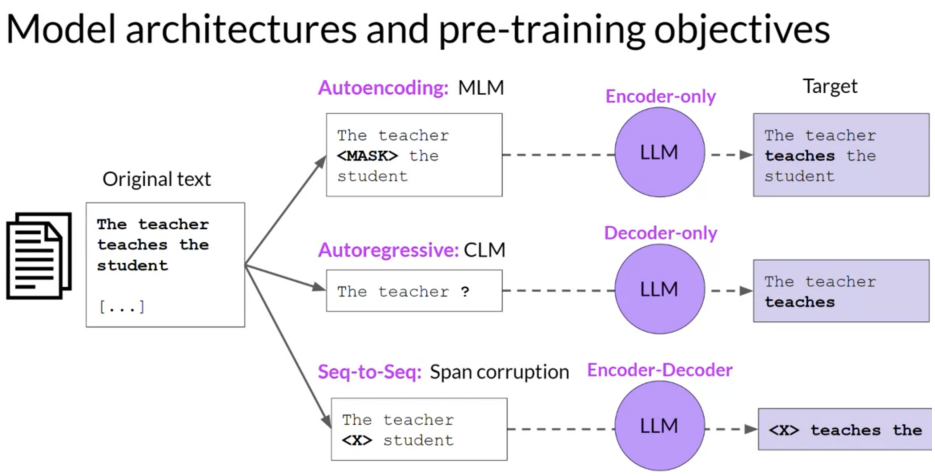
The third type of transformer models are Encoder/Decoder or sequence-to-sequence models. It masks a f few words from the sentence, combines it into one token, then it tries to predict the masked tokens (which is also called sentinel token). These models are good for translation, text summarization, and question answering. BART and T5 are of this type.

Figure 5.8: Different transformer models



Model architectures and pre-training objectives of these three types of transformers are depicted in the next picture.

Figure 5.9: Pretraining objective of each transformer types



Issues of transformers: Hallucinations are words or phrases that are generated by the model that are often nonsensical or grammatically incorrect. This may cause by a number of factors: when a model is not trained on enough data, trained on noisy or dirty data, not given enough context, not

given enough constraints.

5.3 Prompt engineering

A prompt is a short piece of text that is given to a large language model or LLM, as input, and it can be used to control the output of the model in a variety of ways. Prompt design is the process of creating a prompt that is tailored to the specific task that the system is being asked to perform. Prompt engineering is the process of creating a prompt that is designed to improve performance.

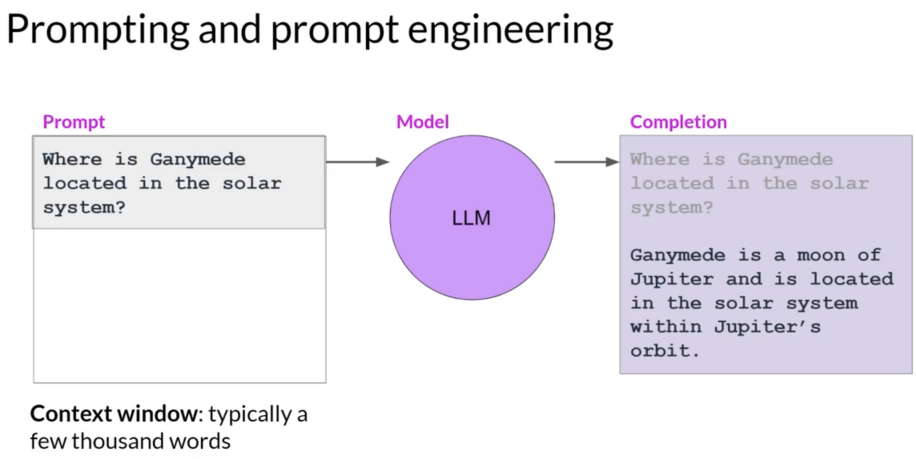
Different Gen AI model types: text-to-text (Ex: translation), text-to-image are trained on a large set of images, each captioned with a short text description, text-to-video, text-to-3D, and text-to-task.

There are three different Language models: Generic (or row) language models: predict the next word (technically, token) based on the language in the training data. Instruction tuned models: trained to predict a response to the instructions given in the input. Dialogue tuned models: trained to have a dialog by predicting the next response.

The more parameter a model has, the more memory it has, and the more sophisticated the task it can perform.

The text you pass to LLM is known as a prompt. The space you have for prompt is called the context window (usually 1000 words). The output of the model is called a completion. The act of using the model to generate an answer is called inference.

Figure 5.10: Different transformer models

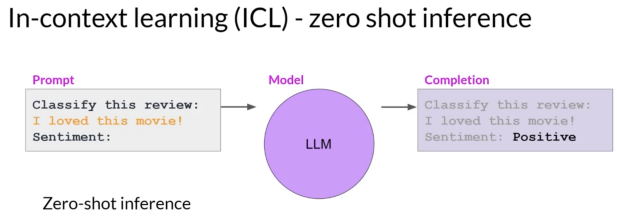


We frequently encounter situations where the model does not produce the outcome you want, on the first try (Zero-shot inference). So, you may have to reviset the language in the prompt.The work to improve the prompt is known as prompt engineering. One way in prompt engineering is to provide examples in the prompt, which is known as in-context learning (ICL). See a one-shot inference in the picture below. We can extend the strategy to have few-shot inference. Sometimes, adding more shots, does not improve the output, for example, if you don't get better answers with 5 shot, probably does not worth try more shots. Also, you need to make sure that you do not exceed the model's input-context length. If the answer is not desirable even with few examples, we need to fine-tune the model.

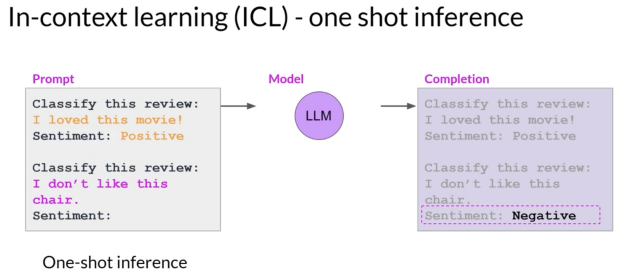
To control LLM behaviors we can set configuration parameters that are different from the training parameters. They invoked at inference time. There are at least four parameters:

1. Max new tokens: Max number of tokens/output the model generate. This is like another stop condition.

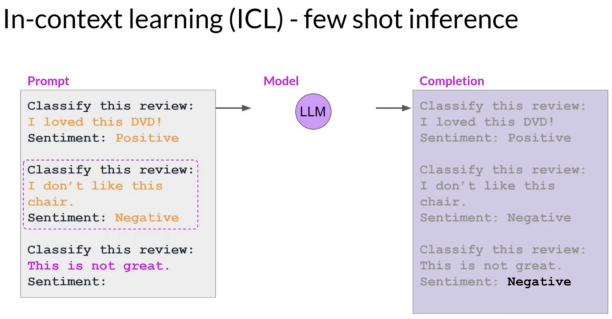
2. Sample top K
   The output from the LLM is a probability distribution across the entire dictionary of words that a model uses. After assigning a probability to each word, there are two modes to select the word, greedy (highest probability is selected) or random sampling (chooses the word from a



(a)



(b)

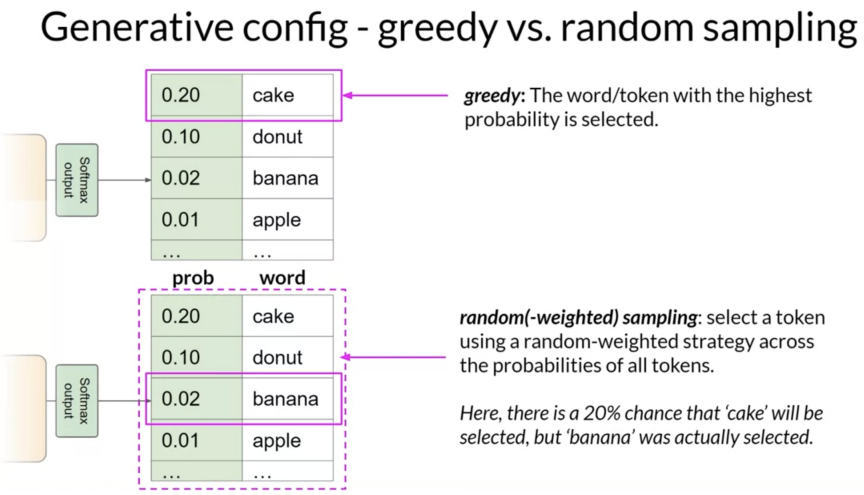


(c)

Figure 5.11: Prompt engineering: (a) zero-shot inference, (b) one-shot inference, (c) few-shot inference

random sampling using a probability distribution.) Greedy method is good for short generation, but it is susceptible to repeated words, or repeated sequences of words. For more natural and creative, and avoid repeating words, we can use random sampling. The output might be too creative to the point that it does not make any sense. So, we have to limit the random sampling. The top-K select an output from the top-k results after applying random-weighted strategy using the probabilities (choosing the from the top K highest probable tokens).

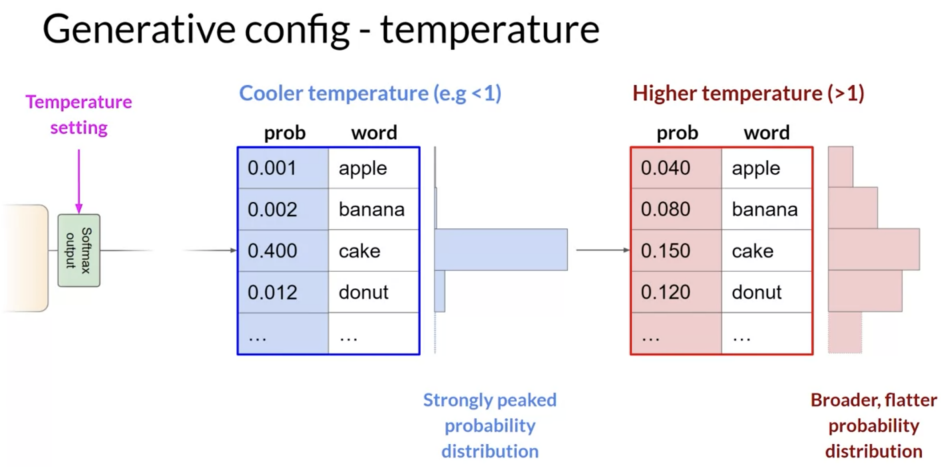
Figure 5.12: Generative configuration



3. Sample top P: Select an output using the random weighted strategy with the top-ranked consecutive results by probability and with a cumulative probability ¡= p.

4. Temperature: impact the shape of probability distribution. The higher the temperature, the higher the randomness.

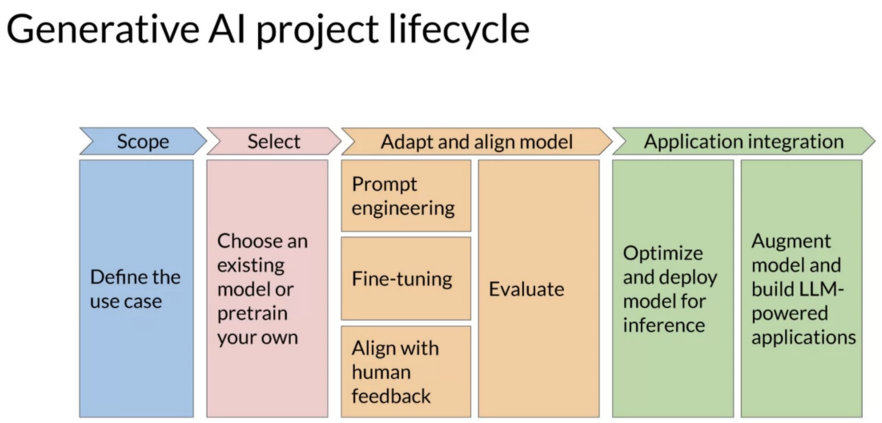
Figure 5.13: Generative configuration



5.4 Project life cycle

After defining the use case, we need to decide whether to use pre-trained LLMs or train your own model. To learn about the pre-trained models, we can refer to model hubs, which includes model cards that describe important details including the best use cases for each model, how it was trained, and known limitations.

Figure 5.14: Generative AI project life cycle



5.5 Quantization

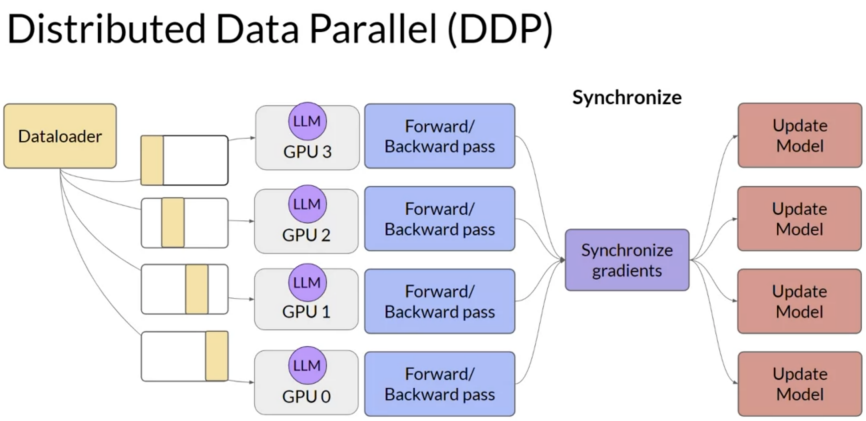
To do the massive computation for training, we need a lot of memory, which might not be available. By quantization, we can use lower precision datatypes, to be able to fit the model on the GPU memory.

5.6 Scale computation

When to use/scale multi-GPU compute strategies: Model too big for single GPU, or, model fits on GPU, but you want to train data in parallel (This is called Distributed Data Parallelism- DDP).

In the DDP, pytorch copies your entire model on all GPUs, and sends batches of Data to each GPU in parallel.

Figure 5.16: Distributed Data Parallel

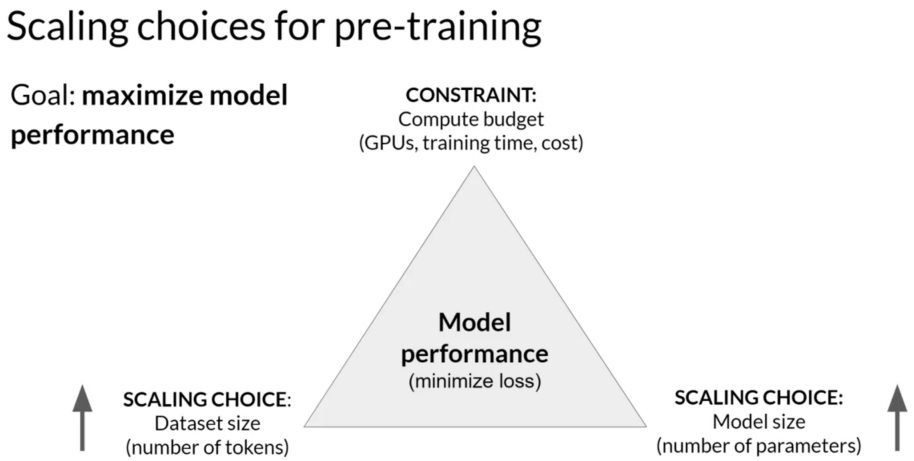


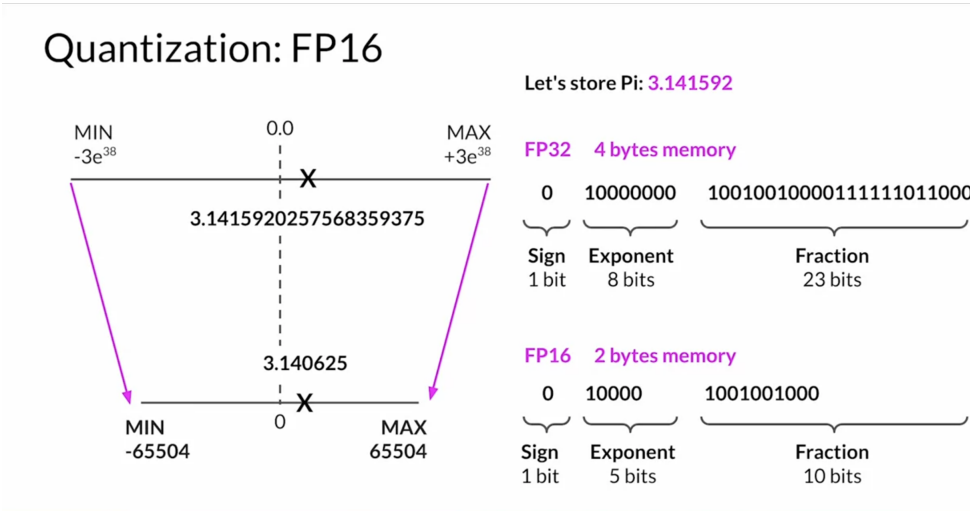
The other option is Fully Sharded Data Parallel (FSDP), motivated by ZeRO (Zero Redundancy Optimizer), which allows to distribute model parameters on multiple GPUs, as opposed to DDP which stores all model parameters on each GPU. ZeRO distribute/shard the model parameters, gradients, and optimizer states across GPUs. It has three stages, depending on which model parameters are distributed. This model allows to work on models that are too big to fit on a single chip. Since not all parameters are available on a GPU, you need to Get weights from all GPUs before you run forward or backward pass. This is more like a memory vs performance trade-off decision. In short, this model, allows you to reduce overall GPU memory utilization, supports offloading to CPU if needed. To manage the trade off between performance and memory utilization, you can configure the level of sharding via sharding factor.

There are three factors that affects the model performance, as depicted in the following picture. The impact of each item on the model performance is linear (in the power loss chart - test loss vs parameter logarithmic chart). What is the ideal balance between these three quantities? According to the Chinchilla paper, the recommended datasize should be 20x of the model parameter size, otherwise, the model has not seen enough data. Based on the results of this paper, many models has not seen enough training data, which means that a smaller model (smaller parameters) with the right amount of data can perform the same as the large models. For this reason, some new

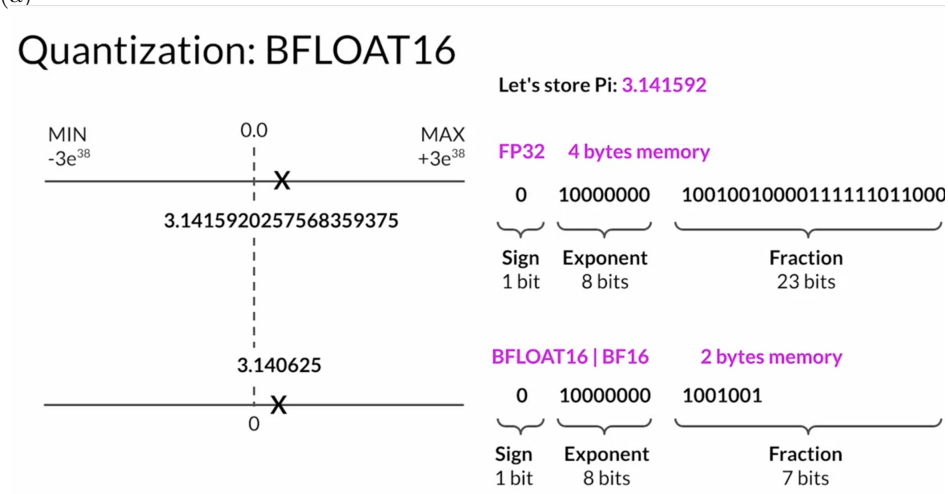
models with smaller number of parameters developed.

Figure 5.18: Distributed Data Parallel



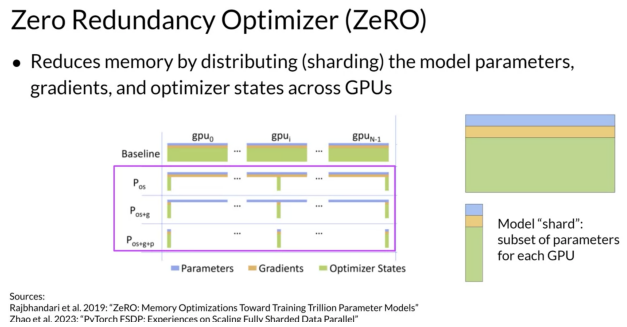


(a)

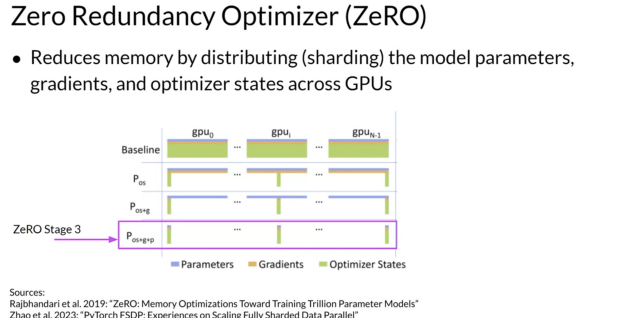


(b)

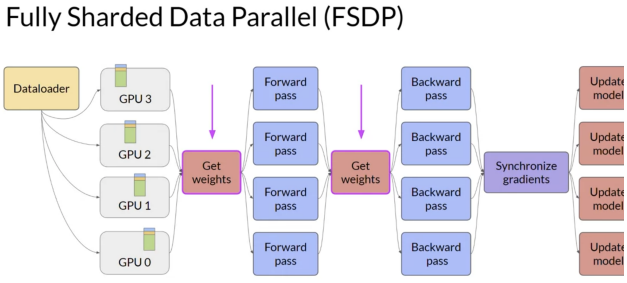
Figure 5.15: Quantization: (a) F16, (b) BF16



(a)



(b)



(c)

Figure 5.17: Multi GPU: (a) Distributing three types of model parameters, (b) Stages of ZeRO, (c) Fully Shared Data Parallel

Chapter 6

Miscellaneous topics

6.1 One hot encoding for categorical features

source1 and source2

A categorical feature takes only a few limited number of values. There are tree options to deal with the categorical features:

- Drop the feature: this option works only if the feature does not provide any useful information.
- Ordinal encoding: assigning a value to each category **(ordinal variables)**. This method works for tree-based (decision tree) models.
- One-hot encoding: creating one feature for each category, and assigning binary values. This method works particularly well if there is no clear ordering in the categorical data. We refer to categorical variables without an intrinsic ranking as **nominal variables**. One-hot encoding generally does not perform well if the categorical variable takes on a large number of values (15 and more). We refer to the number of unique entries of a categorical variable as the cardinality of that categorical variable. High cardinality columns can either be dropped from the dataset, or we can use ordinal encoding.

A one hot encoding is a representation of categorical features as binary vectors. This first requires that the categorical values be mapped to integer values. Then, each integer value is represented as a binary vector that is all zero values except the index of the integer, which is marked with a 1.

Example: assume we have a sequence of labels with the values *red* and *green*. We can assign *red* an integer value of 0 and *green* the integer value of 1. As long as we always assign these numbers to these labels, this is called an integer encoding. Consistency is important so that we can invert the encoding later and get labels back from integer values, such as in the case of making a prediction. Next, we can create a binary vector to represent each integer value. The vector will have a length of 2 for the 2 possible integer values. The *red* label encoded as a 0 will be represented with a binary vector [1, 0] where the zeroth index is marked with a value of 1. In turn, the *green* label encoded as a 1 will be represented with a binary vector [0, 1] where the first index is marked with a value of 1. If we had the sequence: *red, red, green*, we could represent it with the integer encoding: 0, 0, 1. And the one hot encoding of: [1, 0], [1, 0], and [0, 1].

Why Use a One Hot Encoding? A one hot encoding allows the representation of categorical data to be more expressive. Many machine learning algorithms cannot work with categorical data directly. The categories must be converted into numbers. This is required for both input and output variables that are categorical. We could use an integer encoding directly, rescaled where needed. This may work for problems where there is a natural ordinal relationship between the categories, and in turn the integer values, such as labels for temperature *cold, warm, and hot*.

There may be problems when there is no ordinal relationship and allowing the representation to lean on any such relationship might be damaging to learning to solve the problem. An example might be the labels dog and cat

In these cases, we would like to give the network more expressive power to learn a probability-like number for each possible label value. This can help in both making the problem easier for the network to model. When a one hot encoding is used for the output variable, it may offer a more nuanced set of predictions than a single label.

We can use libraries such is scikit-learn and keras to encode a categorical feature for ML.

6.2 Evaluating the model-partitioning the data set

Partition the data into three sets: training set, cross validation set (AKA validation/development/dev set), and test set. Train using the training set, but use the cost function with the cross validation set to evaluate the model (without the regularization parameter). To estimate the generalization error use the test set. The data allocation varies as the number of samples grows. For example, for smaller data set (¡ 10,000) 60%/20%/20% is reasonable, but for very large data set (1,000,000), we can use 98/1/1.

The other way to evaluate a classification model is find the fraction of the test set and the fraction of the train set that the algorithm has mis-classified.

When selecting a model, you want to choose one that performs well both on the training and cross validation set. It implies that it is able to learn the patterns from your training set without overfitting. If you used the defaults in this lab, you will notice a sharp drop in cross validation error from the models with degree=1 to degree=2. This is followed by a relatively flat line up to degree=5. After that, however, the cross validation error is generally getting worse as you add more polynomial features. Given these, you can decide to use the model with the lowest cv_mse as the one best suited for your application.

Mismatched train/test distribution: they should be of the same distribution. Choose the dev set and test set to reflect data you expect to get in the future and consider important to do well on. For example, if you training cat recognition models, and have 200K good images and 10K mobile images (the actual use-case distribution), it is not a good idea to merge all 210K images, shuffle, and split. The better option is to split the 10K mobile data into dev and test sets, each 2.5K, and add 5K mobile images to the good 200K images for training.

For the size of the test set, we need to set it such that it is big enough to give high confidence in the overall performance of your system. For some applications, we don't need to have test set, just train and dev is fine.

6.3 Cross validation method

The goal is always to generalize the model to fit the real world data. The test set is just a representation of the data that we may see in future. The model

should be complex enough to model the training set without over-fitting.

The data points that we are using for training are assumed to be Independent and Identically Distributed (IID), which means that there's no inherent difference between training, testing and real-world data and all the data is coming from the same source. This is the *Fundamental Assumption of Supervised Learning.*

A fixed validation set leaves some random chance in determining model scores. That is, a model might do well on the fixed validation set, even if it would be inaccurate on a different set. The larger the validation set, the less randomness (aka *noise*) there is in our measure of model quality, and the more reliable it will be. Unfortunately, we can only get a large validation set by removing rows from our training data, and smaller training datasets mean worse models! The idea is that we select part of the training set as the test set, not touching the actual test during the training set. This is indeed the cross validation set.

Cross Validation steps:

- Randomly partition the training data into k folds of equal size.
- Train the model on all the folds except for one (k-1).
- Validate the model's performance using the fold that was not used in training.
- Repeat steps 2 and 3 using different combinations of these folds.
- Average the error in predicting the polynomial trained on the training folds.

Cross-validation gives a more accurate measure of model quality, which is especially important if you are making a lot of modeling decisions. However, it can take longer to run, because it estimates multiple models (one for each fold). For small datasets, where extra computational burden isn't a big deal, you should run cross-validation. For larger datasets, a single validation set is sufficient. Your code will run faster, and you may have enough data that there's little need to re-use some of it for holdout. There's no simple threshold for what constitutes a large vs. small dataset. But if your model takes a couple minutes or less to run, it's probably worth switching to cross-validation. Alternatively, you can run cross-validation and see if the scores

for each experiment seem close. If each experiment yields the same results, a single validation set is probably sufficient. Note that we no longer need to keep track of separate training and validation sets.

To use cross validation, it is more convenient to implement a pipeline.

Observation for using cross validation with linear regression:

- Average cross validation error is higher than training error for lower orders .
- Average cross validation error decreases as the polynomial order increases (same for training error).
- After the golden degree, increasing the degree of polynomial will result in overfitting, and the Cross Validation error will start to increase.

6.4 Accuracy of the model: F1 score, precision, and recall

Refer to this source.

Precision (P) is defined as the number of true positives (T_p) over the number of true positives plus the number of false positives (F_p): $P = \frac{T_p}{T_p+F_p}$. This criterion implies the percentage of true positive (of all examples that are positive, what percentage are actually positive). This means if ML detects a class/category, how likely it is correct.

Recall (R) is defined as the number of true positives (T_p) over the number of true positives plus the number of false negatives (F_n): $R = \frac{T_p}{T_p+F_n}$. This criterion implies what percentage of actual positives are correctly recognized. This means if ML can correctly identify the class even if the class is very rare.

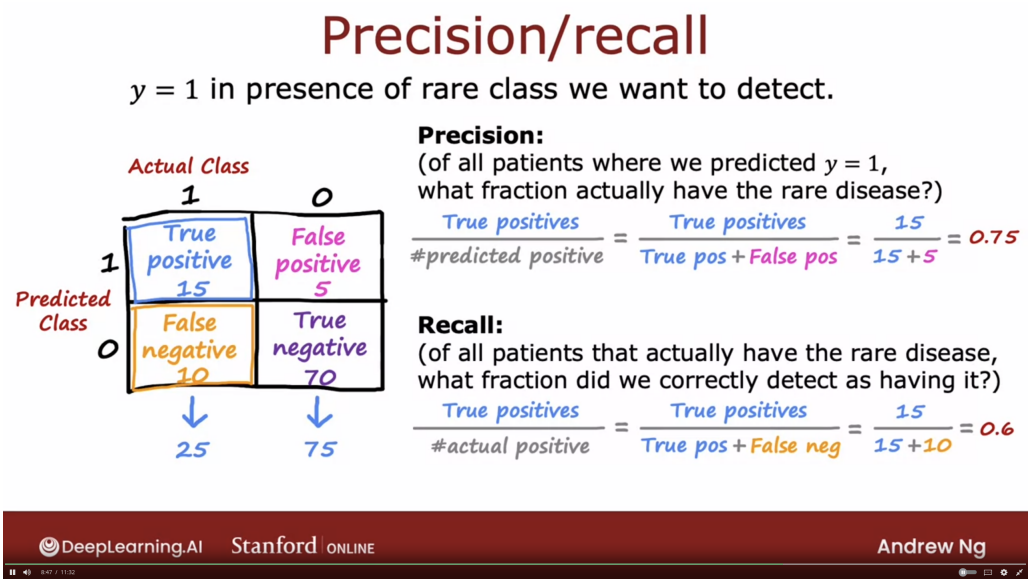


Figure 6.1: confusion matrix

These quantities are also related to the ($F1$) score, which is defined as the harmonic mean of precision and recall: $F1 = 2\frac{R*P}{P+R}$. The F1 score can be interpreted as a harmonic mean of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. F1 score allows us to select the best model among many models with different recall and precision. F1 combines both recall and precision.

The other important factor is the running time of the model. We can combine accuracy and running time into one metric. What we want is to maximize accuracy subject to running time less than a certain amount. In that sense, running time itself becomes an optimization metric. This metric is just a satisficing metric (should be less than a certain amount).

6.5 Optimization

6.5.1 Cost function

Loss function: Used when we refer to the error for a single training example. Cost function: Used to refer to an average of the loss functions over an entire training dataset. The cost function gives you a way to measure the error,

and how well a specific set of parameters fits the training data. Thereby gives you a way to try to choose better parameters. The selection of the loss function depends on the model.

Let's use the concept of multiple linear regression (which is the same for NN) to discuss the cost function.

Parameters/coefficients/weights of the model (In ML, parameters of a model are the variables that you can do training in order to improve the model):

$$w_1, ..., w_n, b \equiv \overrightarrow{w}, b \tag{6.1}$$

Here is the model:

$$f_{\overrightarrow{w},b}(\overrightarrow{x}) = w_1 x_1 + ... + w_n x_n + b \tag{6.2}$$

data points $\{(x_i, y_i) : i = 1...m\}$

$$\hat{y}^{(i)} = f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}) = \overrightarrow{w}\,\overrightarrow{x}^{(i)} + b \tag{6.3}$$

Problem statement: find $\overrightarrow{w}, b$ such that $\hat{y}^{(i)}$ is close to $y^{(i)}$ for all $(x^{(i)}, y^{(i)})$. $\left(\hat{y}^{(i)} - y^{(i)}\right)^2$, the error for one single sample is called the loss function.

To find $\overrightarrow{w}, b$, we form the the cost function. The cost function depends on the method we measure the error. There are many methods to define loss/cost functions. One example is Mean Absolute Error (MAE), which measures the disparity from the true target by an absolute difference: abs(y_true - y_pred).

find $\overrightarrow{w}, b$:

$$\text{minimize: } J(\overrightarrow{w}, b) = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right) \tag{6.4}$$

The other widely used method is **mean squared error cost (MSE)**:

find $\overrightarrow{w}, b$:

$$\text{minimize: } J(\overrightarrow{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{2m} \sum_{i=1}^{m} \left(wx^{(i)} + b - y^{(i)}\right)^2 \tag{6.5}$$

m is the to total number of data points. The extra 2 in the denominator is just for convenience. This is the squared error cost function, which is widely used. In sklearn, we can calculate the cost with mean_squared_error. However, there are other options for the cost functions as well. For example, the squared error cost function is not an ideal cost function for logistic regression, using Sigmoid function. The reason is that the logistic regression model, which is defined by (sigmoid):

$$f_{\vec{w},b}(\overrightarrow{x}) = \frac{1}{1 + e^{-(\vec{w}\,\vec{x}^{(i)} + b)}} \tag{6.6}$$

is non-convex, as opposed to linear regression model which is convex. For logistic/classification problems, we cannot use Mean Absolute Error (MAE) either for the same reason. To have a convergent gradient descent method, the cost function should be convex.

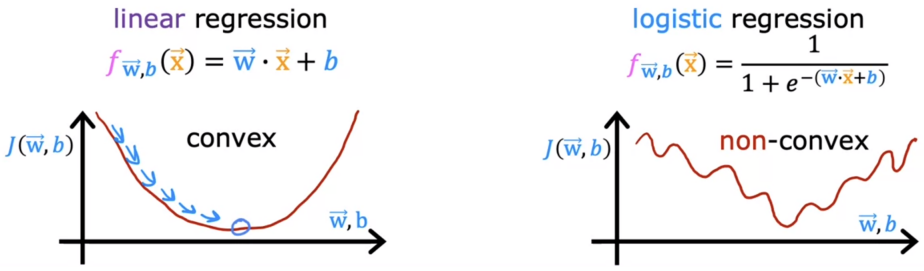


Figure 6.2: cost function for logistic function

To convert the logistic function to a convex function (for a binary classification), we need to ue the following loss definition known as logistic loss.

AKA binary cross entropy (see ML C1W3+ML C2W2). What we want instead is a distance between probabilities, and this is what cross-entropy provides. Cross-entropy is a sort of measure for the distance from one probability distribution to another. The idea is that we want our network to predict the correct class with probability 1.0. The further away the predicted probability is from 1.0, the greater will be the cross-entropy loss:

$$J(\vec{w}, b) = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}L\big(\hat{y}^{(i)} - y^{(i)}\big) = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}L\big(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\big)$$

$$L\big(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\big) = \begin{cases} -log\big(f_{\vec{w},b}(\vec{x}^{(i)})\big) & \text{if } y^{(i)} = 1 \\ -log\big(1 - f_{\vec{w},b}(\vec{x}^{(i)})\big) & \text{if } y^{(i)} = 0 \end{cases} \tag{6.7}$$

If $y^{(i)} = 1$ then, if the predicted value $f_{\vec{w},b}(\vec{x}^{(i)})$ is close to one, the loss function is close to zero (reduces the loss), but if the predicted value is close to zero, then the function penalize the loss function with a large number. The other way around for $y^{(i)} = 0$.

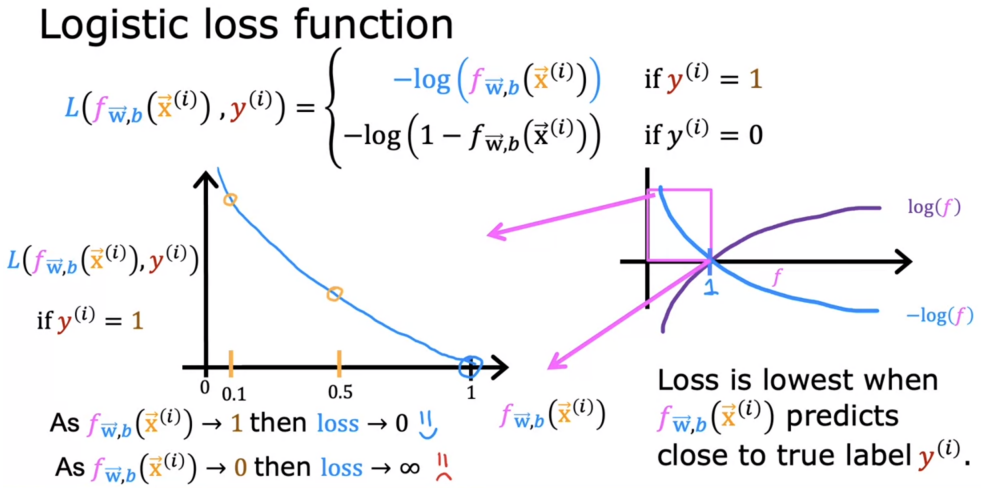


Figure 6.3: cost function for logistic function

To simplify this loss and cost functions, we re-write it as follows:

$$L\big(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}) - y^{(i)}\big) = -y^{(i)}log\big(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)})\big) - (1-y^{(i)})log\big(1-f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)})\big)$$
$$J(\overrightarrow{w},b) = -\frac{1}{m}\sum_{i=1}^{m}\Big[y^{(i)}log\big(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)})\big) + (1-y^{(i)})log\big(1-f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)})\big)\Big]$$

(6.8)

Here is the summary of the steps for training any ML model (nn, linear regression, etc.-see code C2W1_NN/2_neuralNet.ipynb):

1. *Define model* by specify how to compute output given input x and parameters w, and b ($f_{\overrightarrow{w},b}(\overrightarrow{x})$).
2. Specify *Loss function* $L(f_{\overrightarrow{w},b}(\overrightarrow{x}))$ and *cost function* $J(\overrightarrow{w},b) = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}L\big(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}) - y^{(i)}\big)$
3. *Train* on data to minimize $J(\overrightarrow{w},b)$, for example using gradient descent methods and back propagation.

The same way, the loss function for softmax is:

$$L\big(a_1,a_2,...,a_m,y\big) = \begin{cases} -log(a_1) \text{ if } y=1 \\ -log(a_2) \text{ if } y=2 \\ \vdots \\ -log(a_m) \text{ if } y=m \end{cases} \tag{6.9}$$

Note in 6.9 above, only the line that corresponds to the target contributes to the loss, other lines are zero. To write the cost equation we need an 'indicator function' that will be 1 when the index matches the target and zero otherwise.

$$\mathbf{1}\{y == n\} == \begin{cases} 1, & \text{if } y == n. \\ 0, & \text{otherwise.} \end{cases}$$

The cost is (called Sparse Ctegorical Cross Entropy)-SparseCtegoricalCrossentropy):

$$J(\mathbf{w},b) = -\left[\sum_{i=1}^{m}\sum_{j=1}^{N} 1\left\{y^{(i)} == j\right\}\log\frac{e^{z_j^{(i)}}}{\sum_{k=1}^{N}e^{z_k^{(i)}}}\right] \tag{4}$$

where m is the number of examples, N is the number of outputs. This is the average of all the losses.

We can also add a weight to this error classification ($\mathbf{1}$) to penalize some cases. For example, if we are training a cat classification model, and the model shows pornographic pics as cats, we can penalize the pornographic images with a larger weight. In that case, the error/cost would be:

$$J = \frac{1}{\sum_{i=1}^{m_{dev}} \mathbf{w^{(i)}}} \sum_{i=1}^{m_{dev}} \mathbf{w^{(i)}} 1\{y^{(i)}_{predic} \neq y^{(i)}\} \tag{6.10}$$

$$\mathbf{w^{(i)}} = \begin{cases} 1, & \text{if} x^{(i)} \text{is non-porn} \\ 10, & \text{if} x^{(i)} \text{is porn} \end{cases} \tag{6.11}$$

The error model/cost function is orthogonal (separate problem) than the training and doing well on the metric itself. First, we need to establish the error/target, then we need to train the model to reduce the error.

For more accurate (less floating point error) calculation in tensorflow, it is preferred to have a linear activation function for the last layer, instead of a softmax (the output is this form call logits) use from_logits=True. This informs the loss function that the softmax operation should be included in the loss calculation. This allows for an optimized implementation. Notice that in the preferred model, the outputs are not probabilities, but the output for each data sample is a vector of size the number of neurons in the last year. This vector for each sample can range from large negative numbers to large positive numbers. The output must be sent through a softmax when performing a prediction that expects a probability. However, we really don't need to apply a softmax to select the most likely category, we can just find the index of the largest output using np.argmax(). See 1_MachineLearning_Stanford_Projects_python_C2W2_2_softmax.ipynb.

6.5.2 Gradient descent (steepest descent) method

An optimization method to minimize any function. Depending on the initial starting point, the method may end up at a different local minimum.

Method:
For a cost function $J(\overrightarrow{w}, b)$, we want to find $\overrightarrow{w}(w_1, w_2, ...), b$ to minimize J.

Algorithm:

- start with some initial $\overrightarrow{w}, b$.
- update w_j *simultaneously* until the algorithm converges, meaning you reach a local minimum, where additional iteration does not change the point. It is incorrect to use new values of w_j (j number of features) to update other parameters in the same iteration.

$$\begin{aligned} w_j =& w_j - \alpha \frac{\partial}{\partial w_j} J(\overrightarrow{w}, b) \\ b =& b - \alpha \frac{\partial}{\partial b} J(\overrightarrow{w}, b) \end{aligned} \tag{6.12}$$

The negative is because we want to move in the opposite direction. α is the learning rate (step size), see 6.5.3.

The derivatives of cost functions for regular regression and logistic regression are:

$$\begin{aligned} \frac{\partial}{\partial w_j} J(\overrightarrow{w}, b) =& \frac{1}{m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right) x_j^{(i)} = \frac{1}{m} \sum_{i=1}^{m} \left(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}) - y^{(i)}\right) x_j^{(i)} \quad \text{for each j} \\ \frac{\partial}{\partial b} J(\overrightarrow{w}, b) =& \frac{1}{m} \sum_{i=1}^{m} \left(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}) - y^{(i)}\right) \end{aligned} \tag{6.13}$$

But note that the definition of f for the two regressions are different (for logistic, f is the sigmoid function).

Back propagation is used to find the derivatives of cost function with respect to its parameters. This method is nothing but the chain rule. This method is cheaper (#layers + #parameter) compared to a direct calculation of the derivatives (#layers $\times$ #parameter). The idea is to use the chain rule from left to right.

Note that since the slope is close to zero at both ends of Sigmoid function, the gradient descent becomes very slow, but with ReLU, we don't have this issue.

Vanishing /Exploding Gradients: For a deep NN, if activation units or weights are greater than one, they explode after multiplying in the forwared propagation. Conversely, if the weights are less than one, they vanish.

This causes a very small training process/gradient descent. To solve this issue, we need to be very careful how we should initialize the weights, by setting the weight, not too much greater or smaller than one.

Local optima in neural network most neural network problems are saddle points rather than local optima. Problem is really not the local optima, but the plateaus, a region where the derivative is close to zero for a long time.

Various types of gradient descent:

Batch gradient descent: each step of gradient descent uses all the training examples. While this batching provides computation efficiency, it can still have a long processing time for large training datasets as it still needs to store all of the data into memory. Batch gradient descent also usually produces a stable error gradient and convergence (if you plot cost vs iteration, ideally it should decrease as the number of iteration increases), but sometimes that convergence point isn't the most ideal, finding the local minimum versus the global one.

Mini-batch gradient descent updates the parameters using a subset of training samples. That means you do a full forward and backward propagation using a subset of samples. If you plot cost vs iterations, it is more noisy but the trend should be the same as a batch gradient descent. The noise is due to the subsets, one subset might have a higher cost, and the other might have a lower cost. The size of the mini-batch determines the type of gradient descent:

- size = m: Batch gradient descent (The cons is that it is long per iteration.)
- size = 1: stochastic gradient descent (every example is its own mini-batch). This one is very noisy, but the average direction is toward the minimum, but may never hit the minimum.
- $1 < size < m($ 1000): This is the fastest learning. It makes progress without processing all training samples. Typical batch size is powers of two (64, 128, 256, 512, and 1024), depending on the memory size of GPU/CPU. The mini-batch size is another hyperparameter.

If the training set is small (less than 2000), go with the batch gradient method.

Epoch is a single pass through the training set. With Batch gradient descent, we do one step of the gradient descent in each epoch, but with the mini-batch gradient descent, we do as many as the number of mini batched gradient descent in each epoch.

Gradient descent with momentum: It moves faster than the regular gradient descent. First, we need to learn about Exponentially weighted average, as defined below to find the average of a quantity, such as temperature.

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t \tag{6.14}$$

V_t is approximately average over $\frac{1}{1-\beta}$. The main advantage of this method, is that it takes a little memory, compared to a moving average that requires more memory (but it is more accurate).

The first few samples of the weighted average is not valid, because we don't have information for the prior points. We need to use bias computation to correct this. Bias correction makes the computation more accurately. The correction is:

$$V_t = \frac{V_t}{1-\beta^t} \tag{6.15}$$

Now that we know what exponential weighted average is, we need to use it with gradient descent to boost the efficiency. on iteration t, compute the dw and db as usual in the backward propagation, then:

$$V_{dw} = V_{db} = 0 \tag{6.16}$$
$$V_{dw} = \beta V_{dw} + (1-\beta)dw \tag{6.17}$$
$$V_{db} = \beta V_{db} + (1-\beta)db \tag{6.18}$$
$$W = W - \alpha V_{dw} \tag{6.19}$$
$$b = b - \alpha V_{db} \tag{6.20}$$
$$\tag{6.21}$$

The β is another hyperparameter in our model, but, in practice, it is ok to skip bias correction. Additionally, in some literature, the term $1-\beta$ is omitted.

RMSprop (Root Mean Square propagation):

on iteration t, compute the dw and db as usual in the backward propagation, then (squared in the following equation is element-wise):

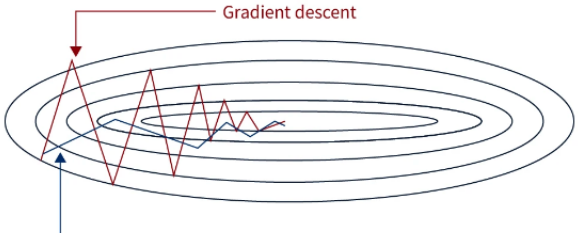


Figure 6.4: Gradient descent with momentum

$$S_{dw} = S_{db} = 0 \tag{6.22}$$
$$S_{dw} = \beta S_{dw} + (1-\beta)dw^2 \tag{6.23}$$
$$S_{db} = \beta S_{db} + (1-\beta)db^2 \tag{6.24}$$
$$W = W - \alpha \frac{dw}{\sqrt{S_{dw}}} \tag{6.25}$$
$$b = b - \alpha \frac{db}{\sqrt{S_{db}}} \tag{6.26}$$
$$\tag{6.27}$$

Adam algorithm for optimization Adam (Adaptive Moment estimation) is the same as the gradient descent method, but it adjusts the learning rate (increases or decreases the rate). In practice, this method uses different learning rates for different parameters. Basically, it is a combination of RMSprop and momentum.

The idea is that if gradient descent direction is roughly the same in each iteration, it increases the learning rate, but if direction is changing, the Adam algorithm reduces the learning rate.

on iteration t, compute the dw and db as usual in the backward propa-

gation, then (squared in the following equation is element-wise):

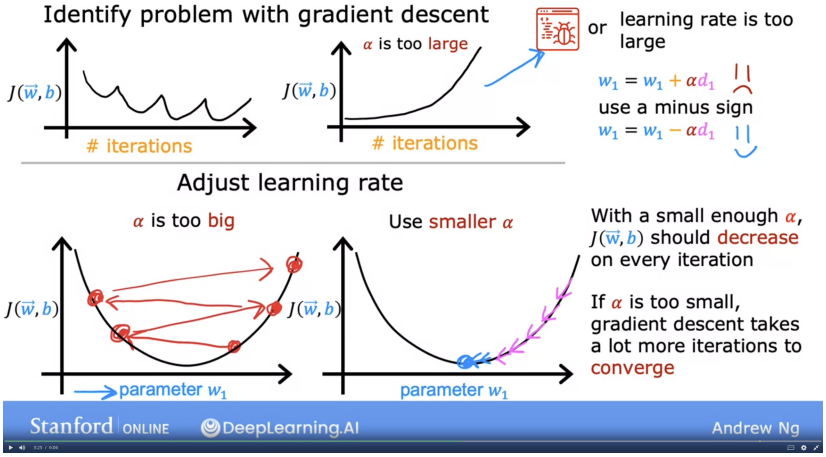
$$S_{dw} = S_{db} = V_{dw} = V_{db} = 0 \tag{6.28}$$
$$V_{dw} = \beta_1 V_{dw} + (1 - \beta_1) dw \tag{6.29}$$
$$V_{db} = \beta_1 V_{db} + (1 - \beta_1) db \tag{6.30}$$
$$S_{dw} = \beta_2 S_{dw} + (1 - \beta_2) dw^2 \tag{6.31}$$
$$S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2 \tag{6.32}$$
$$V_{dw}^{corrected} = \frac{V_{dw}}{(1 - \beta_1^t)} \tag{6.33}$$
$$V_{db}^{corrected} = \frac{V_{db}}{(1 - \beta_1^t)} \tag{6.34}$$
$$S_{dw}^{corrected} = \frac{S_{dw}}{(1 - \beta_1^t)} \tag{6.35}$$
$$S_{db}^{corrected} = \frac{S_{db}}{(1 - \beta_1^t)} \tag{6.36}$$
$$W = W - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected} + \epsilon}} \tag{6.37}$$
$$b = b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected} + \epsilon}} \tag{6.38}$$
$$\tag{6.39}$$
$$\tag{6.40}$$

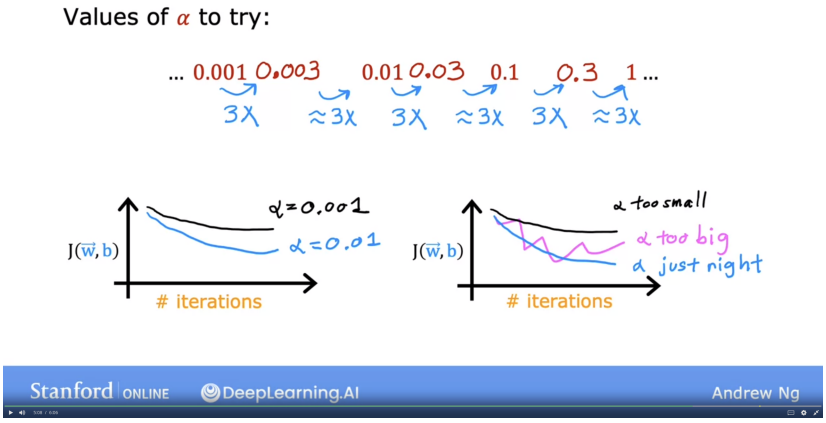
Hyperparameters are: $\alpha, \beta_1(0.9), \beta_2(0.999), \epsilon(10^{-8})$.

6.5.3 Learning rate selection

The choice of learning rate affects the efficiency of gradient descent. If the learning rate is too small, it would be very slow, and if it too large (overshoot), it may diverge. To start, we can set the learning rate to a very small number, and check if the cost function decreases with every single iteration. If not, there is bug in the code. We can plot the cost function (learning curve) with each iteration of the gradient descent. The learning rate α should be selected such that the cost function decreases as iterations increases. If this is not the case, then the learning rate is high. Generally, we need to test many learning rates and select the most optimized one.

If we are exactly at the minimum, since slope is zero, the value of the learning does not affect the results. Thus, even a fixed learning can reach the minimum.





Learning rate decay: To speed up the learning algorithm, we can slowly reduce the learning rate over iterations. This helps with the mini-batched gradient descent, which potentially, has a noisy nature, and the algorithm may never reach to the minimum. By reducing the learning rate, we can reduce the region in which the algorithm is oscillating. Different formulas are used in the literature:

$$\alpha = \frac{1}{1 + decayRate \times epochNumber}\alpha_0 \tag{6.41}$$

$$\alpha = 0.95^{epochNumbre}\alpha_0 \text{ (exponential decay)} \tag{6.42}$$

$$\alpha = \frac{k}{\sqrt{epochNum}}\alpha_0 \tag{6.43}$$

staircase decay (6.44)

Manual decay (6.45)

Decay rate is another hyperparameter.

6.6 Overfitting (high variance) vs underfitting (high bias)

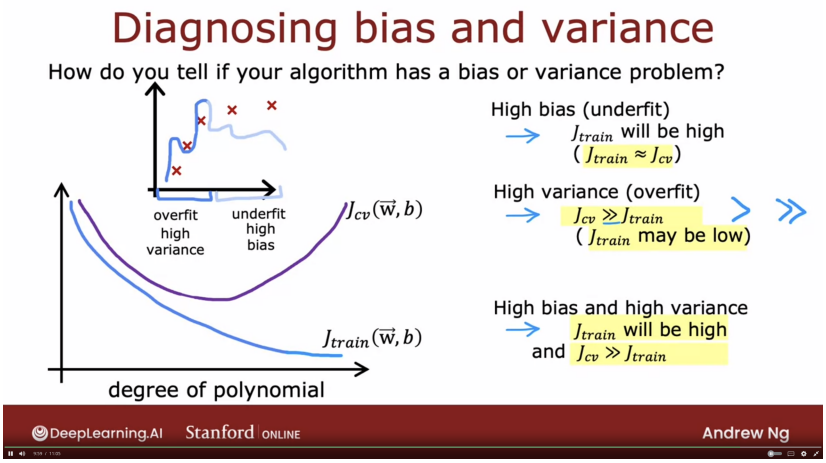
Overfitting (high variance) is the case that a model matches the training data almost perfectly, i.e. J_{train} is low, but does poorly in validation and other new data, i.e. J_{cv} is high. On the flip side, When a model fails to capture important distinctions and patterns in the data, so it performs poorly even in training data, i.e. J_{train} and J_{cv} are high, that is called underfitting (high bias). Note that bias has a other meanings, such as checking the models based on characteristics such as gender, ethnicity, etc.

We can think of overfitting/underfitting as follows. the information in the training data as being of two kinds: signal and noise. The signal is the part that generalizes, the part that can help our model make predictions from new data. The noise is that part that is only true of the training data; the noise is all of the random fluctuation that comes from data in the real-world or all of the incidental, non-informative patterns that can't actually help the model make predictions. The noise is the part might look useful but really isn't. Now, the training loss will go down either when the model learns signal or when it learns noise. But the validation loss will go down only when the model learns signal. (Whatever noise the model learned from the training set won't generalize to new data.) So, when a model learns signal both curves go down, but when it learns noise a gap is created in the curves. The size of the gap tells you how much noise the model has learned. Underfitting the training set is when the loss is not as low as it could be because the model hasn't learned enough signal. Overfitting the training set is when the loss

is not as low as it could be because the model learned too much noise. The trick to training deep learning models is finding the best balance between the two.

The case in the middle is "just right" model where J_{train} and J_{cv} are low.

Figure 6.5: Bias vs Variance



For some neural network applications, there are some cases where the model has the high bias and high variance at the same time.

Since we care about accuracy on new data, which we estimate from our validation data, we want to find the sweet spot between underfitting and overfitting, so that the model do well on examples that are not in the training set. This is called *generalization*.

To avoid underfitting:

- adding new features: fixes high bias
- adding polynomial features/bigger neural network fixes high bias (Wider networks have an easier time learning more linear relationships, while deeper networks prefer more nonlinear ones. Which is better just depends on the dataset.)
- decreasing the regularization (*lambda*) fixes high bias

To avoid overfitting:

- one method is to collect more training data. If this option is available, probably we should try this first, but this is not always the option.

- feature selection: review features, and use fewer features. The down side of this method is that we are throwing away some useful information about the model. See 6.11.
- try getting additional features
- try adding polynomial features x_1^2, x_2^2, etc.
- increasing regularization ($lambda$) fixes high variance. regularization: it gently reduces the impact of some features, without eliminating the feature. This, in some sort, means that we can have higher order parameters in the model, without overfitting. See 6.8
- Early stopping: the gap between the cross validation (dev) and training error decreases, until it grows at some point. We can stop once the gap between the dev and training error increases.

To add data, we can use the augmentation of data (rotating images, cropping images, distorting images, etc.). Data augmentation does not add a lot of information to our data set, but in case we really cannot get more data, we can use this method. If there is not enough data to train a model, one option is to use the ***transfer learning***, in which you train a model using data from a different task (this step is called spervised pretraning), and fine tune the model with the actual data (this step is called fine tuning.). The other option is to use the other model parameters, and use the actual data to train only the last year of the neural network.

One more important topic is the baseline level of performance. In other words, how should one know if the J_{train} is low, or high, or good enough. Once method to find the baseline performance is to compare the error with respect to the human-level performance. The other options are competing algorithm performance, or guess based on the past experience.

Note also that the best possible error is called Bayes optimal error. There is not a big difference between human-level and Bayes optimal error, thus, we can use the human-level error as a proxy for the Bayes error. In general, the training process is pretty fast, between until it reaches the human-level error, but slows down between the human-level and the Bayes. Knowing the human-level error, allows us to better analyze bias and variance.

If there is a gap between the baseline and the training error, then model in underfit (high bias), but if the gap is between the train and cross validation, it is the overfit (high variance) problem.

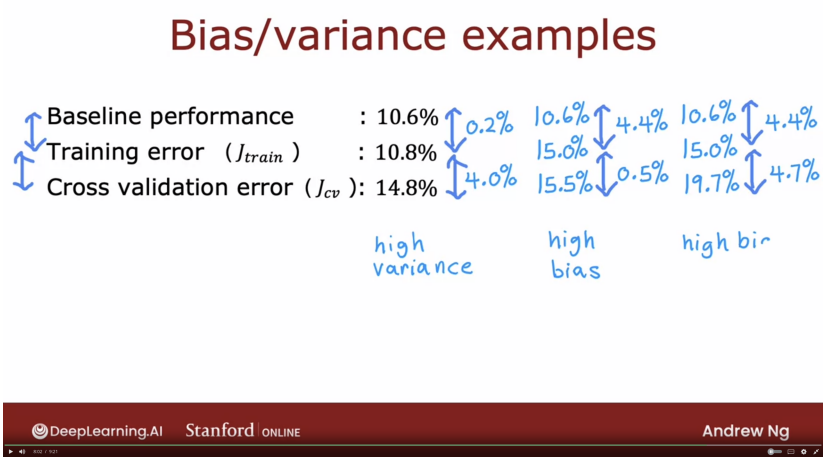


Figure 6.6: Bias vs Variance

All the above argument is valid if the train and dev sets are from the same distribution. But if dev and training are from a different distribution. For example, if there is a gap between training and dev error (usually considered as high variance problem), the gap might be due to the difference in the dev set (dev set has more challenging samples). We mentioned in 6.2 that the it is more efficient if the dev set contains samples from the real application (ex. mobile data), in that case, dev set has more difficult samples, compared to the training set which has high quality images/samples. The high dev set error is due to two items, one is the model has not seen this set before (new data which is the source of high variance), or second, the distribution gap issue (data mismatch problem). To detect the high dev set error is from which of these cases, we can define a new set call training-dev set, with the same distribution as the training set, but you did not trained on that. In short, we have four sets, one: train, two: train-dev (both from the same distribution), three: dev, and four: test (3 and 4 from the same distribution but different from the first two sets). To do the error analysis, we get the error for all four sets. If there is a gap between train and train-dev (but little gap between train-dev and dev), this is an indication of the high variance problem. On the other hand, if train-dev error is only a bit above train, but has a good gap with the dev set, then the issue is data mismatch.

The other case is that if training, training-dev, dev, and test sets are close, but much higher than the human-level, this is an indication of the high bias problem.

Another note, it may also happen that dev/test set are much easier than the train and train-dev sets. In those cases, the error on dev/test might be even lower than the error in the train set.

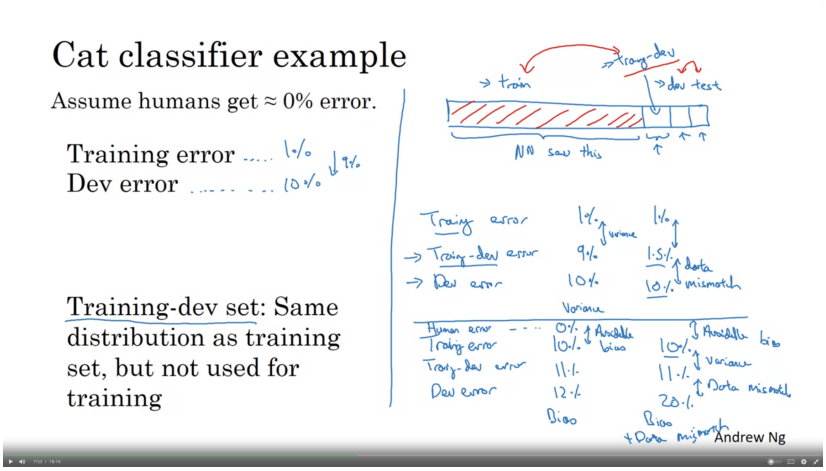


Figure 6.7: Bias vs Variance distribution

Here is the chart to decide how to adjust the model:

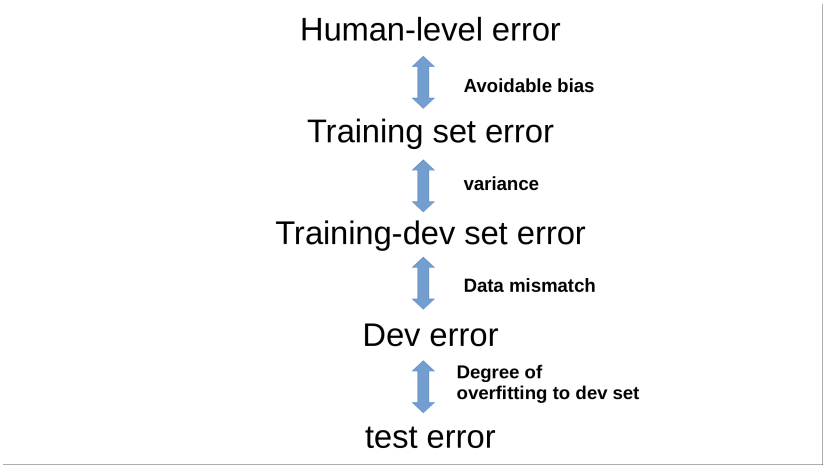


Figure 6.8: Bias vs Variance model based on error analysis

To resolve a data mismatch, e can carry out manual error analysis to try to understand difference between training and dev/text sets. The other

option is to make training data more similar or collect more data similar to dev/test sets. You can synthesis artificial data.

Note also that as the training set size increases, J_{train} increases, because it is harder to fit the curve, but generally J_{cv} drops.

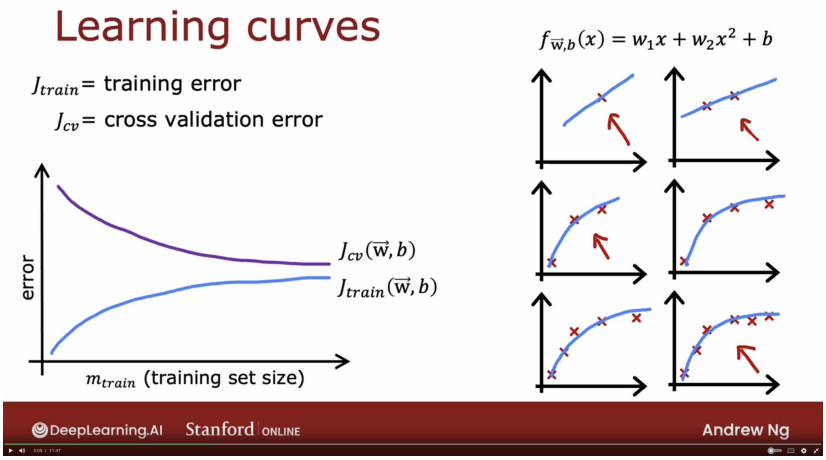


Figure 6.9: learning curve

For a model with high bias, the addition of sampling sizes after certain point, does not help with the performance.

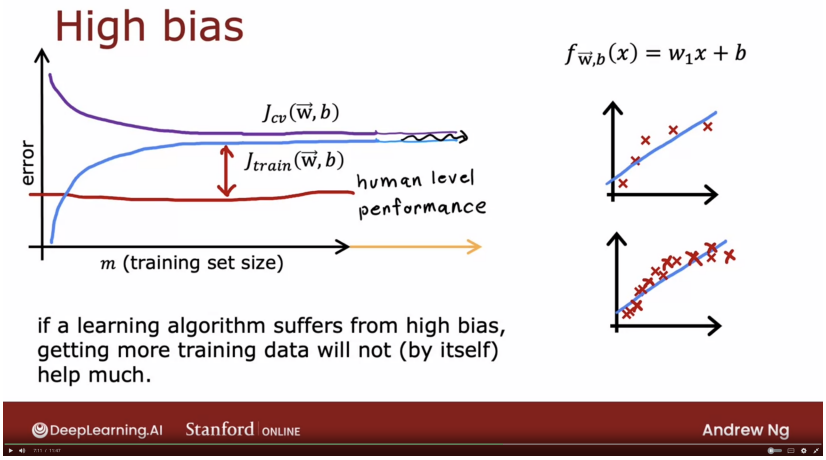


Figure 6.10: High bias

For a model with high variance, increase the training set results in converging the errors to the baseline level.

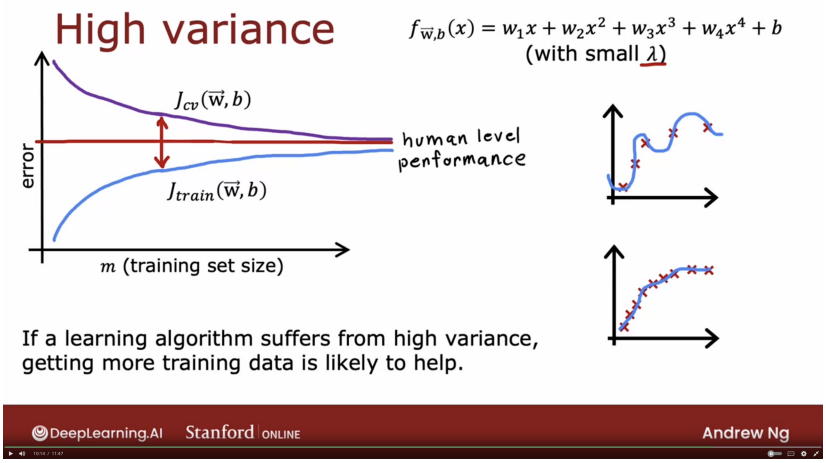


Figure 6.11: High variance

How does a neural network work in terms of bias and variance? Large neural networks are low bias machines, means, they generally fit the model. Using a bigger nn (more hidden layers or more units per layer) reduces the bias. A large neural network will usually do as well or better than a smaller one so long as requlariza tion is chosen appropriately. A larger neural network does not hurt, just more computationally intense. With neural network, the main problem is high variance rather than high bias.

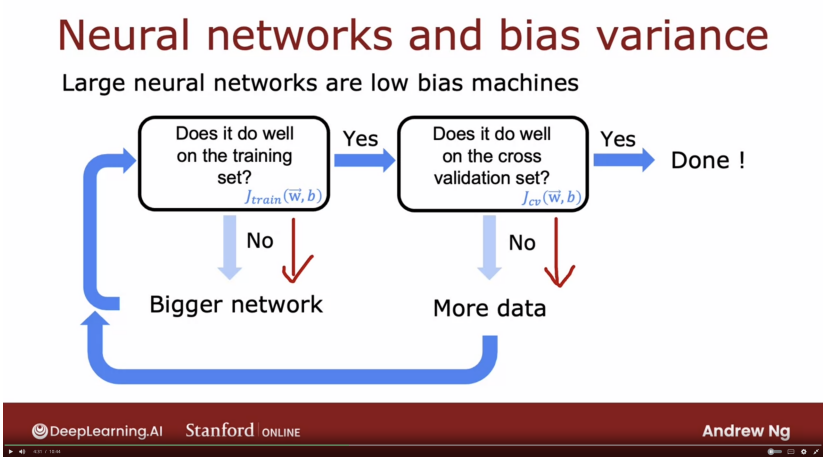


Figure 6.12: Neural network training

The other topic is orthogonalization. For training, we try to optimize the cost function, but at the same time, we want to avoid overfitting. These two items are orthogonal, and more optimization means more overfitting. Instead, we need a tool to mix these two goals.

6.7 Transfer learning and multi-task learning

It is the idea of taking the knowledge the neural network has learned from one task and apply that knowledge to a separate task. Based on this method, you train a network based on a data set, then, you re-train the last layer or even adding more layers, using the new data set.

Transfer learning makes sense when 1) both tasks/data have the same input, 2) when you have a lot of data for the data set you are transferring from, and less data for the set you are transferring to (It does not make sense for the other case (less data for the first set, more data for the second set)), and 3) low level features from the frist set coulld be helpful for learning the second set.

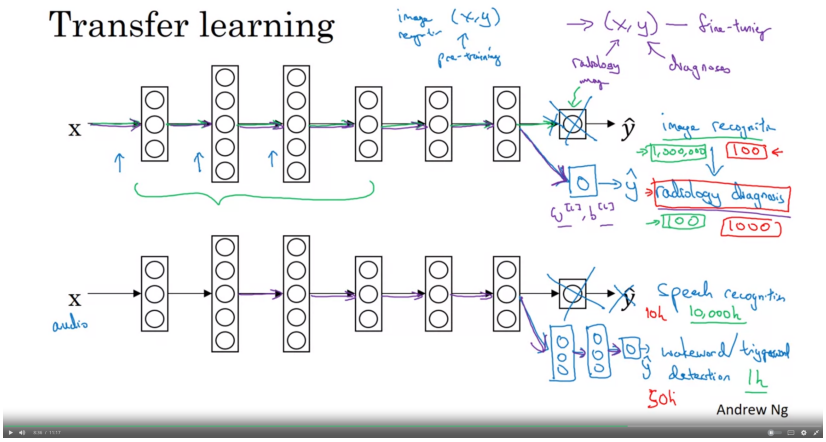


Figure 6.13: Transfer learning

In multi task learning, you start training an nn doing multiple tasks at the same time. In this manner, the final output layer has a dimension for all tasks. For example, if you are training an nn, to detect cars, pedestrians, and signs, then, the output layer should have three units, each for one class.

Multi-task learning makes sense when 1) training on a se of tasks that could benefit from having shared lower-level features, 2) amount of data you have for each task is quite similar.

6.8 Regularization

Lambda method We need to modify the cost function by penalizing/regularize the features/parameters corresponding to over fitting. When there are many features for a model, it is prone to overfit. To avoid that, the idea is that to build a model with all features, because initially we don't know which features are important, but we penalize all features.

For a linear regression, we get (we are not penalizing b param, if we do it looks like this):

$$J(\overrightarrow{w}, b) = \underbrace{\frac{1}{2m}\sum_{i=1}^{m}\left(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}) - y^{(i)}\right)^2}_{\text{squared mean error}} + \underbrace{\frac{\lambda}{2m}\sum_{i=1}^{n} w_j^2 + \frac{\lambda}{2m}b^2}_{\text{regularization term}} \tag{6.46}$$

This is the $L_2(||w||_2^2)$ regularization, but we can also use the $L_1(||w||_1)$ regularization.

The regularization for the neural network is applied to all weights:

$$J(w^{[1]}, b^{[1]}, ..., w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} L\big(\hat{y}^{(i)} - y^{(i)}\big) + \underbrace{\frac{\lambda}{2m}\sum_{i=1}^{l} ||w^{[l]}||_F^2}_{\text{regularization term}} \tag{6.47}$$

where, the Frobenius norm is:

$$||w^{[l]}||_F^2 = \sum_{i=1}\sum_{j=1}(w_{ij}^{[l]})^2 \;\; w(n^{[l]}n^{[l-1]}) \tag{6.48}$$

The regularization term forces the coefficient to remain small and λ is the coefficient to control this. A large λ results in underfit by reducing weights, but a very small one, leads to overfit. To find the right value for the regularization parameter, we can train the model with a sweep of λ s and calculate J_{cv} for each case, and pick a λ, corresponding to the lowest cost function. But this requires running the model many times for each regularization parameter.

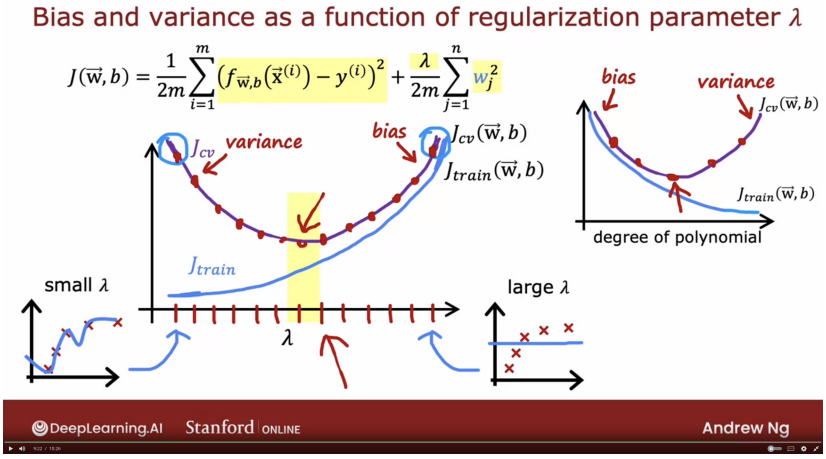


Figure 6.14: Regularization parameter

Gradient descent with cost function (without b), looks like this (rewriting

equation 6.13):

$$\begin{aligned}\frac{\partial}{\partial w_j}J(\overrightarrow{w},b) =& \frac{1}{m}\sum_{i=1}^{m}\left(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}) - y^{(i)}\right)x_j^{(i)} + \overbrace{\frac{\lambda}{m}w_j}^{\text{regularization term}} \qquad \text{for each j}\\ \frac{\partial}{\partial b}J(\overrightarrow{w},b) =& \frac{1}{m}\sum_{i=1}^{m}\left(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}) - y^{(i)}\right)\end{aligned} \tag{6.49}$$

For a logistic regression, we get:

$$J(\overrightarrow{w},b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}log\left(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)})\right)+(1-y^{(i)})log\left(1-f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)})\right)\right]+ \underbrace{\frac{\lambda}{2m}\sum_{i=1}^{n}w_j^2}_{\text{regularization term}} \tag{6.50}$$

The derivatives of equation 6.50 for gradient descent is similar to 6.49, just f is different.

Dropout regularization:

We set a probability to drop units in each layer of neural network. For each sample then we use the reduce network. To implement dropout, we can use the inverted dropout method. In each iteration, we randomly drop some units. There would be no dropout for predictions

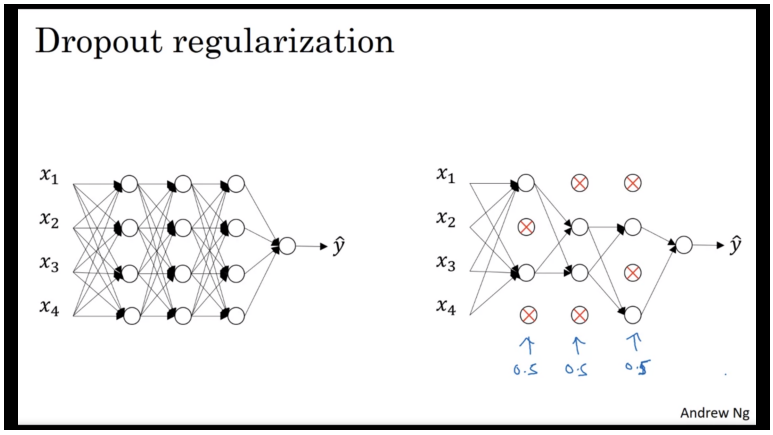


Figure 6.15: Neural network training

6.9 Model validation

Always start with the dataset documentation.

In most (though not all) applications, the relevant measure of model quality is predictive accuracy. In other words, will the model's predictions be close to what actually happens.

There are many metrics for summarizing model quality, but we'll start with one called Mean Absolute Error (also called MAE). With the MAE metric, we take the absolute value of each error. This converts each error to a positive number. We then take the average of those absolute errors. This is our measure of model quality. sklearn has a function, mean_absolute_error, to calculate MAE.

Many people make a mistake when measuring predictive accuracy. They make predictions with their training data and compare those predictions to the target values in the training data. This is not valid. The most straightforward way to validate the model is to exclude some data from the model-building process, and then use those to test the model's accuracy on data it hasn't seen before. This data is called validation data.

6.10 Data processing/cleaning: Missing Values

Before we treat missing data, we need to know **Is this value missing because it wasn't recorded or because it doesn't exist?** If a value is missing because it doesn't exist (like the height of the oldest child of someone who doesn't have any children) then it doesn't make sense to try and guess what it might be. These values you probably do want to keep as NaN. On the other hand, if a value is missing because it wasn't recorded, then you can try to guess what it might have been based on the other values in that column and row. There are many ways the data misses: wrong measurement, data is private, data is not available, etc.

Generally, there are three approaches deal with the missing data. These are quick and dirty approaches that probably remove some useful info or adding some noise to your dataset.

- The simplest option: drop columns or features with missing values. Unless most values in the dropped columns are missing, the model loses

access to a lot of (potentially useful!) information with this approach.

- Imputation: fills in the missing values with some number. For instance, we can fill in the mean value along each column, or we can choose the most popular data for the missing item. We can also assign the probability of each value of the attribute to the missing entries. The imputed value won't be exactly right in most cases, but it usually leads to more accurate models than you would get from dropping the column entirely.

- Extending imputation: we impute the missing values, as before. And, additionally, for each column with missing entries in the original dataset, we add a new column that shows the location of the imputed entries. For example, if the size of bedroom is missing for some instances, we add a new column, missing_bedromm_size, and set it to FALSE for all valid instances and TRUE for all missing instances. This will meaningfully improve results. In other cases, it doesn't help at all.

In pandas, we can use .isnul() to detect columns with missing data. We can use .drop to drop columns. We can also use dropna to either drop columns or instances.

6.11 Data processing & Feature engineering

6.11.1 Feature engineering

The goal of feature engineering is simply to make your data better suited to the problem at hand. You might perform feature engineering to:

- improve a model's predictive performance
- reduce computational or data needs
- improve interpretability of the results

Feature engineering is possible through:

- determine which features are the most important with mutual information

- invent new features in several real-world problem domains (e.g. square the length of land, to get area, which is a more reasonable feature. It can predicate parabola now.)
- encode high-cardinality categoricals with a target encoding
- create segmentation features with k-means clustering
- decompose a dataset's variation into features with principal component analysis

A great first step is to construct a ranking with a feature utility metric, a function measuring associations between a feature and the target. Then you can choose a smaller set of the most useful features to develop initially and have more confidence that your time will be well spent. The metric we'll use is called "mutual information". It can **detect any kind of relationship, while correlation only detects linear relationships.** The mutual information (MI) between two quantities is a measure of the extent to which knowledge of one quantity reduces uncertainty about the other. Mutual Info varies between 0 (independent quantities) to $+\infty$. It is a logarithmic func, and barely reaches to 2. It's possible for a feature to be very informative when interacting with other features, but not so informative all alone. MI can't detect interactions between features. It is a univariate metric. The actual usefulness of a feature depends on the model you use it with. A feature is only useful to the extent that its relationship with the target is one your model can learn.

Scikit-learn has two mutual information metrics in its feature_selection module: one for real-valued targets (mutual_info_regression) and one for categorical targets (mutual_info_classification).

6.11.2 How to create new features

Mathematical transform

Use arithmetic operations to create new features based on other features. For example, find the area of land by multiplying width by length. This will create non-linearity in the date.

For the neural network, you don't need to create non-linear features. Neural networks can learn non-linear relationships between the inputs and

outputs. But for linear regression, you need to create non-linear features to capture non-linear relationships between the inputs and outputs.

Count

Features describing the presence or absence of something often come in sets, the set of risk factors for a disease, say. You can aggregate such features by creating a count. These features will be binary (1 for Present, 0 for Absent) or boolean (True or False). In Python, booleans can be added up just as if they were integers.

Example:
accidents["RoadwayFeatures"] = accidents[roadwayFeatures].sum(axis=1)

Building up and breaking down features

Add or breakdown strings to create new features. For example, break down phone numbers to area codes.

Clustering

Use, for example, K-means to cluster data points and add it as a feature. It requires a knowledge about the domain.

Principal Component Analysis (PCA)

There are two ways you could use PCA for feature engineering.

- The first way is to use it as a descriptive technique. Since the components tell you about the variation, you could compute the MI scores for the components and see what kind of variation is most predictive of your target. That could give you ideas for kinds of features to create.
- The second way is to use the components themselves as features. Because the components expose the variational structure of the data directly, they can often be more informative than the original features. Here are some use-cases:
  - Dimensionality reduction: When your features are highly redundant (multicollinear, specifically), PCA will partition out the redundancy into one or more near-zero variance components, which you can then drop since they will contain little or no information.

- Anomaly detection: Unusual variation, not apparent from the original features, will often show up in the low-variance components. These components could be highly informative in an anomaly or outlier detection task.
- Noise reduction: A collection of sensor readings will often share some common background noise. PCA can sometimes collect the (informative) signal into a smaller number of features while leaving the noise alone, thus boosting the signal-to-noise ratio.
- Decorrelation: Some ML algorithms struggle with highly-correlated features. PCA transforms correlated features into uncorrelated components, which could be easier for your algorithm to work with.

PCA basically gives you direct access to the correlational structure of your data. You'll no doubt come up with applications of your own!

PCA Best Practices- there are a few things to keep in mind when applying PCA:

- PCA only works with numeric features, like continuous quantities or counts.
- PCA is sensitive to scale. It's good practice to standardize your data before applying PCA, unless you know you have good reason not to.
- Consider removing or constraining outliers, since they can have an undue influence on the results.

Target Encoding

This method, as opposed to all other methods, targets categorical features. It is similar to one-hot or label encoding, with the difference that it also uses the target to create the encoding. This makes it what we call a supervised feature engineering technique.

A target encoding is any kind of encoding that replaces a feature's categories with some number derived from the target. For example, *mean encoding*, takes the average of the numerical target based on a categorical feature (ex. average/mean price of a car brand (make)). **Bin counting** is the equivalent encoded feature applied to a binary target.

Target encoding is great for:

- High-cardinality features: A feature with a large number of categories can be troublesome to encode: a one-hot encoding would generate too many features and alternatives, like a label encoding, might not be appropriate for that feature. A target encoding derives numbers for the categories using the feature's most important property: its relationship with the target.

- Domain-motivated features: From prior experience, you might suspect that a categorical feature should be important even if it scored poorly with a feature metric. A target encoding can help reveal a feature's true informativeness.

There are at least two concerns with the target encoding:

- unknown categories: Target encodings create a special risk of overfitting, which means they need to be trained on an independent "encoding" split. When you join the encoding to future splits, Pandas will fill in missing values for any categories not present in the encoding split. These missing values you would have to impute somehow.

- rare categories: When a category only occurs a few times in the dataset, any statistics calculated on its group are unlikely to be very accurate.

A solution to these problems are **smoothing**. The idea is to blend the in-category average with the overall average. Rare categories get less weight on their category average, while missing categories just get the overall average.

In pseudocode: encoding = weight * in_category (ex. mean of the category for that feature) + (1 - weight) * overall (ex. mean of all categories for that feature). Where weight is a value between 0 and 1 calculated from the category frequency. An easy way to determine the value for weight is to compute an m-estimate: weight = n / (n + m), where n is the total number of times that category occurs in the data. The parameter m determines the "smoothing factor". Larger values of m put more weight on the overall estimate. When choosing a value for m, consider how noisy you expect the categories to be. Does the price of a vehicle vary a great deal within each make? Would you need a lot of data to get good estimates? If so, it could be better to choose a larger value for m; if the average price for each make were relatively stable, a smaller value could be okay.

6.11.3 Scaling vs normalization

In both cases, you're transforming the values of numeric variables so that the transformed data points have specific helpful properties to speed of the training. The point is to help with the gradient descent to converge faster. The difference is that: in **scaling**, you're changing the range of your data, while in **normalization**, you're changing the shape of the distribution of your data.

scaling

This means that you're transforming your data so that it fits within a specific scale, like 0-100 or 0-1. You want to scale data when you're using methods based on measures of how far apart data points are, like support vector machines (SVM) or k-nearest neighbors (KNN). With these algorithms, a change of "1" in any numeric feature is given the same importance. By scaling your variables, you can help compare different variables on equal footing.

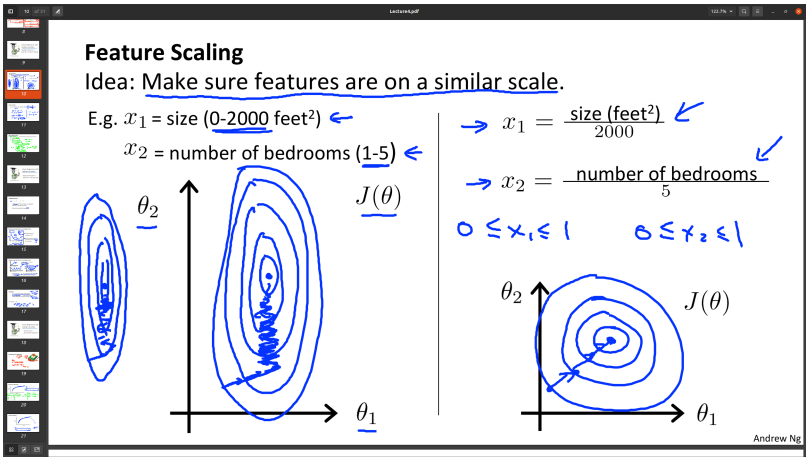
normalization

Scaling just changes the range of your data. Normalization is a more radical transformation. The point of normalization is to change your observations so that they can be described as a normal distribution.

In general, you'll normalize your data if you're going to be using a machine learning or statistics technique that assumes your data is normally distributed. Some examples of these include linear discriminant analysis (LDA) and Gaussian naive Bayes. (Pro tip: any method with "Gaussian" in the name probably assumes normality.)

6.11.4 Scaling a feature

Having two features with large and small values may have adverse impact on our training. For example, for regression, it may cause large round off error, or affects the shape of gradient descent contours and as a result may cause slow convergence or even divergence. To avoid that, it is recommended to scale features so that they have the same range. After scaling, the data range would be 0 to 1.



6.11.5 Mean normalization

In this method, we first find the average of the feature (μ). Each item of the feature is normalized as:

$$x_i = \frac{x_i - \mu}{max - min} \tag{6.51}$$

Then, we have the range for each feature between -1 and 1.

6.11.6 z-score normalization

In this method, we first find the standard deviation (σ) and the average (μ) of the feature. Each item of the feature is normalized as:

$$x_i = \frac{x_i - \mu}{\sigma} \tag{6.52}$$

Then, we have the range for each feature between -1 and 1. If you partition the set to training, cross validation, and testing sets, and z-score normalize training, you need to normalize testing and cross validation sets with the mean and variance of the training set to ensure that your input features are transformed as expected by the model.

6.11.7 Combining features

Using intuition to design new features by transforming or combining original features.

6.11.8 Batch normalization

This is the normalization of the activation units $a^{[l-1]}$ to help optimizing $W^{[l]}$ and $b^{[l]}$ in the next layer more efficiently. In the literature, we can apply normalization on $Z^{[l-1]}$ (before applying the activation function) or on $a^{[l-1]}$ (after applying the activation function). The former is more often. The normalization equation is the same as z-score normalization. ϵ is to avoid division by zero.

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{6.53}$$

We don't want all the hidden units have the mean zero and variance one to take advantage of the non-linearity of the model, especially with the sigmoid function, so to avoid that, we adjust the equation as follows:

$$\tilde{Z}^{(i)} = \gamma z^{(i)}_{norm} + \beta \tag{6.54}$$

where γ and β are learnable parameters (not hyperparameters) that can be optimized in each iteration, for example, using gradient descent.

This is how the batch norm is applied in a neural network:

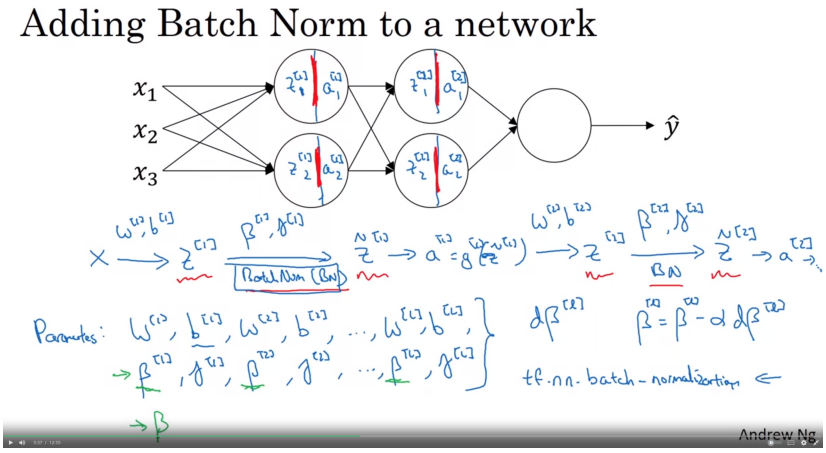


Figure 6.16: Batch normalization

If we use batch norm with mini-batch, we can eliminate the b parameter, because it will be zeroed by the batch norm process anyways.

Here is the implementation of the nn with gradient descent (similarly for GD with momentum, RMSprop, Adam) with batch norm:

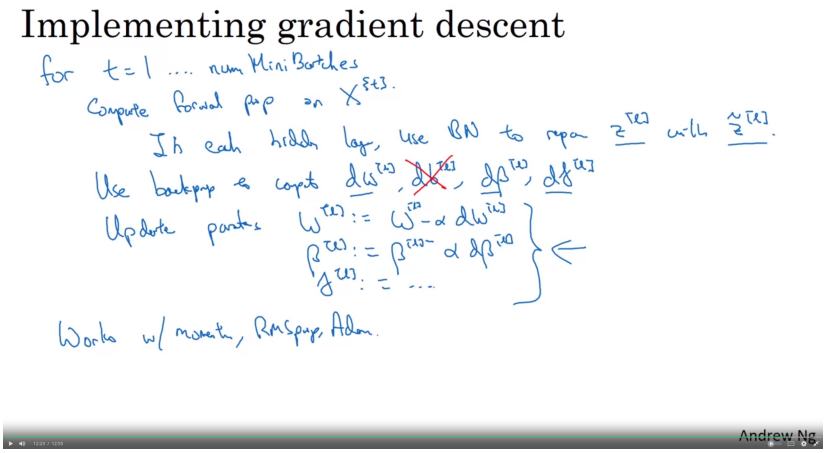


Figure 6.17: NN with Batch normalization

One other impact that batch norm has in the training of a neural network is that it makes weights later or deeper in the nn more robust to changes to weights in earlier layers of the neural network. This is called covariant shift. This has some regularization effect.

6.12 Ensemble methods

Ensemble methods combine the predictions of several models (e.g., several trees, in the case of random forests).

6.12.1 Gradient Boosting

Gradient boosting is a method that goes through cycles to iteratively add models into an ensemble.

It begins by initializing the ensemble with a single model, whose predictions can be pretty naive. (Even if its predictions are wildly inaccurate, subsequent additions to the ensemble will address those errors.)

Then, we start the cycle:

- First, we use the current ensemble to generate predictions for each observation in the dataset. To make a prediction, we add the predictions from all models in the ensemble.
- These predictions are used to calculate a loss function (like mean squared error, for instance).
- Then, we use the loss function to fit a new model that will be added to the ensemble. Specifically, we determine model parameters so that adding this new model to the ensemble will reduce the loss. (Side note: The "gradient" in "gradient boosting" refers to the fact that we'll use gradient descent on the loss function to determine the parameters in this new model.)
- Finally, we add the new model to ensemble,
- repeat!

XGBoost stands for extreme gradient boosting, which is an implementation of gradient boosting with several additional features focused on performance and speed. (Scikit-learn has another version of gradient boosting, but XGBoost has some technical advantages.) scikit-learn API for XGBoost (xgboost.XGBRegressor) allows us to build and fit a model just as we would in scikit-learn. The XGBRegressor class has many tunable parameters. See the code.

6.12.2 Data leakage

Data leakage (or leakage) happens when your training data contains information about the target, but similar data will not be available when the model is used for prediction. This leads to high performance on the training set (and possibly even the validation data), but the model will perform poorly in production.

In other words, leakage causes a model to look accurate until you start making decisions with the model, and then the model becomes very inaccurate.

There are two main types of leakage: target leakage and train-test contamination.

Target leakage occurs when your predictors include data that will not be available at the time you make predictions. It is important to think about

target leakage in terms of the timing or chronological order that data becomes available, not merely whether a feature helps make good predictions.

Example: People take antibiotic medicines after getting pneumonia in order to recover. The raw data shows a strong relationship between those columns, but *took antibiotic medicine* is frequently changed after the value for *got pneumonia* is determined. This is target leakage. Since validation data comes from the same source as training data, the pattern will repeat itself in validation, and the model will have great validation (or cross-validation) scores. But the model will be very inaccurate when subsequently deployed in the real world, because even patients who will get pneumonia won't have received antibiotics yet when we need to make predictions about their future health.

To prevent this type of data leakage, any variable updated (or created) after the target value is realized should be excluded.

Train-Test Contamination A different type of leak occurs when you aren't careful to distinguish training data from validation data.

Recall that validation is meant to be a measure of how the model does on data that it hasn't considered before. You can corrupt this process in subtle ways if the validation data affects the preprocessing behavior. This is sometimes called train-test contamination.

For example, imagine you run preprocessing (like fitting an imputer for missing values) before calling train_test_split(). The end result? Your model may get good validation scores, giving you great confidence in it, but perform poorly when you deploy it to make decisions.

After all, you incorporated data from the validation or test data into how you make predictions, so the may do well on that particular data even if it can't generalize to new data. This problem becomes even more subtle (and more dangerous) when you do more complex feature engineering.

As we discussed in the tutorial, to avoid overfitting, we need to fit the encoder on data heldout from the training set.

If your validation is based on a simple train-test split, exclude the validation data from any type of fitting, including the fitting of preprocessing steps. This is easier if you use scikit-learn pipelines. When using cross-validation, it's even more critical that you do your preprocessing inside the pipeline!

6.13 Python

- Use vectorization in python to improve the performance (for example: np.dot(W,x)).
- To expand the dimension of a numpy array: np.expand_dims(x, axis=1). This will convert (dim,) to (dim,1). For the reverse action, i.e. converting (dim,1) to (dim,), array[:,0].

6.14 ML Checklist

- Inspect all features, look for duplicate features.
- Manage missing data (drop items with missing target, and impute missing items).
- Manage categorical/object features (cordinal, one-hot, etc.)

6.14.1 Time series

- Plot lags to find seasonality/cycles and calculate correlation, and add new lag features if there is a high correlation.
- Plot partial autocorrelation, and add new lag features if there is a high correlation.

6.15 Time series

The basic object of forecasting is the time series, which is a set of observations recorded over time. In forecasting applications, the observations are typically recorded with a regular frequency, like daily or monthly.

One of the main forecasting approaches is the Linear regression method for forecasting (target = weight_1 * feature_1 + weight_2 * feature_2 + bias). But, first we need to generate the features corresponding to the time series. There are three kinds of features unique to time series: time-step features and lag features.

Time-step features are features we can derive directly from the time index. The most basic time-step feature is the time dummy, which counts

off time steps in the series from beginning to end. Time-step features let you model time dependence. A series is time dependent if its values can be predicted from the time they occurred.

Lag (and lead) features are the second types of features that we can generate from the time series. To make a lag feature, we shift the observations of the target series so that they appear to have occurred later in time. Lag features let you model *serial dependence*. A time series has serial dependence when an observation can be predicted from previous observations. **Adapting machine learning algorithms to time series problems is largely about feature engineering with the time index and lags.**

Seasonal features such as seasonal *indicators* or Fourier transform features (will be discussed later).

There are three patterns/components of dependence in a time series:

Series = **trend** (moving average or linear regression (with different polynomial order) with time-step feature) + **seasonality** (weekly, monthly, etc. using indicator features (for short seasons) or Fourier method (if season is longer)) + **cycles** (serial dependence using lag feature) + **error**. Each of these items is called a component.

The residuals of a model are the difference between the target the model was trained on and the predictions the model makes – the difference between the actual curve and the fitted curve, in other words.

Trend, seasonality, and cycles will be discussed next.

6.15.1 Trend

This portion of the time series can be captured by moving average rolling windows, or using linear regressions with the time-step feature. The trend models we learned about in this lesson turn out to be useful for a number of reasons. Besides acting as a baseline or starting point for more sophisticated models, we can also use them as a component in a "hybrid model" with algorithms unable to learn trends (like XGBoost and random forests).

6.15.2 Seasonality

We say that a time series exhibits seasonality whenever there is a regular, periodic change in the mean of the series. Seasonal changes generally follow the clock and calendar – repetitions over a day, a week, or a year are common.

There are two kinds of features that model seasonality. The first kind, *indicators*, is best for a season with few observations, like a weekly season of daily observations (7 observation in the season). The second kind, *Fourier features*, is best for a season with many observations, like an annual season of daily observations (365 observations per season).

we can use a seasonal plot to discover seasonal patterns. A seasonal plot shows segments of the time series plotted against some common period, the period being the "season" you want to observe.

Seasonal indicators

A seasonal plot shows segments of the time series plotted against some common period, the period being the "season" you want to observe. Seasonal indicators are binary features that represent seasonal differences in the level of a time series. Seasonal indicators are what you get if you treat a seasonal period as a categorical feature and apply one-hot encoding. For example, by one-hot encoding days of the week, we get weekly seasonal indicators. Creating weekly indicators for the Trigonometry series will then give us six new "dummy" features. (Linear regression works best if you drop one of the indicators; we chose Monday in the frame below. This means the in the one-hot model, if all Tue Sun is 0, it is Monday.)

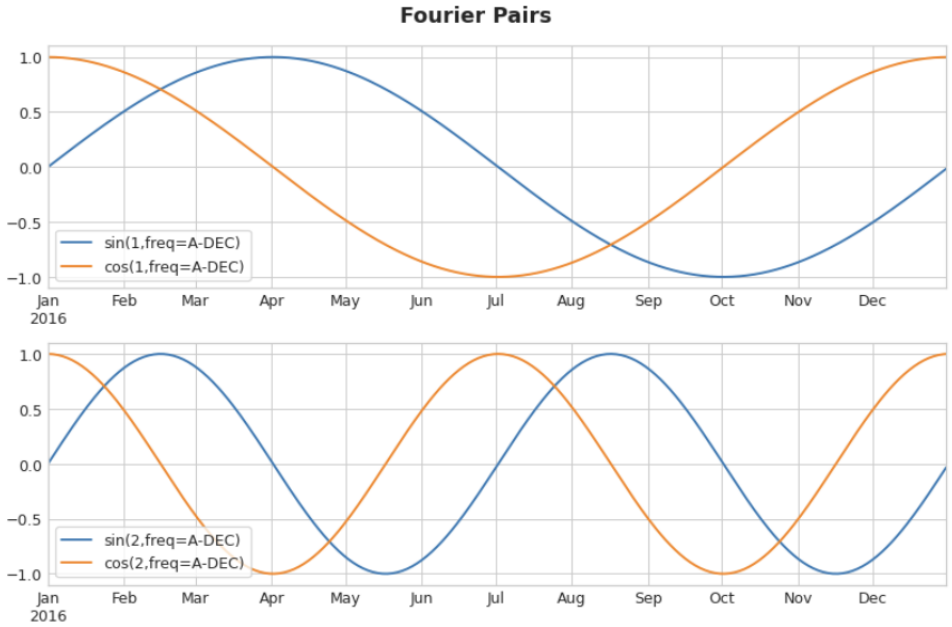
the *Trigonometry* series will then give us six new "dummy" features. (Linear regression works best if you drop one of the indicators; we chose Monday in the frame below.)

Date	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
2016-01-04	0.0	0.0	0.0	0.0	0.0	0.0
2016-01-05	1.0	0.0	0.0	0.0	0.0	0.0
2016-01-06	0.0	1.0	0.0	0.0	0.0	0.0
2016-01-07	0.0	0.0	1.0	0.0	0.0	0.0
2016-01-08	0.0	0.0	0.0	1.0	0.0	0.0
2016-01-09	0.0	0.0	0.0	0.0	1.0	0.0
2016-01-10	0.0	0.0	0.0	0.0	0.0	1.0
2016-01-11	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...

Figure 6.18: seasonal indicators

Fourier features and periodogram

The kind of feature we discuss now are better suited for long seasons over many observations where indicators would be impractical. Instead of creating a feature for each date, Fourier features try to capture the overall shape of the seasonal curve with just a few features. It is these frequencies within a season that we attempt to capture with Fourier features. The idea is to include in our training data periodic curves having the same frequencies as the season we are trying to model. Fourier features are pairs of sine and cosine curves, one pair for each potential frequency in the season starting with the longest. Fourier pairs modeling annual seasonality would have frequencies: once per year, twice per year, three times per year, and so on.

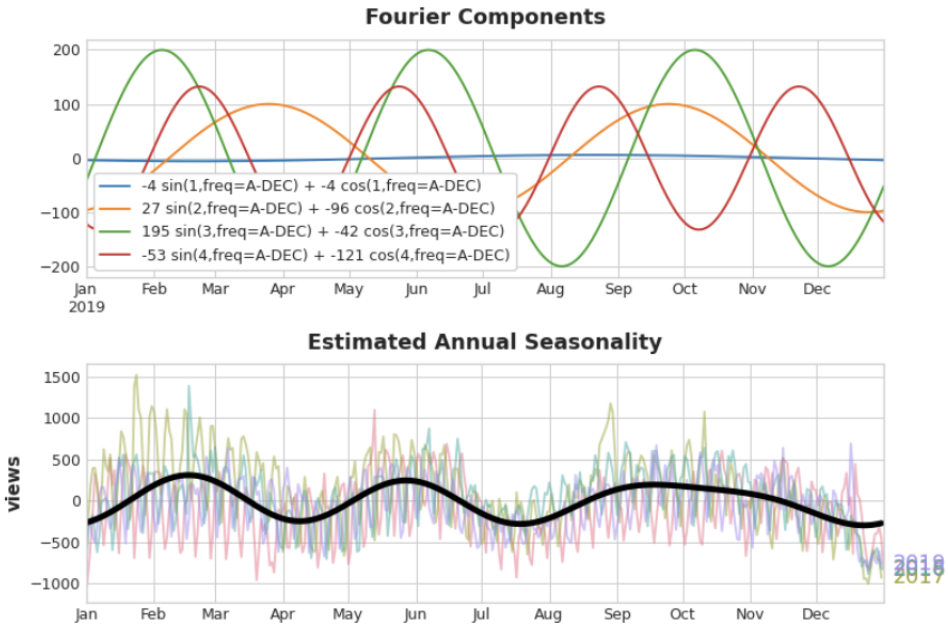


The first two Fourier pairs for annual seasonality. ***Top:*** *Frequency of once per year.* ***Bottom:*** *Frequency of twice per year.*

Figure 6.19: Fourier features

If we add a set of these sine / cosine curves to our training data, the linear regression algorithm will figure out the weights that will fit the seasonal com-

ponent in the target series. The figure illustrates how linear regression used four Fourier pairs to model the annual seasonality in the Wiki Trigonometry series.



Top: *Curves for four Fourier pairs, a sum of sine and cosine with regression coefficients. Each curve models a different frequency.* ***Bottom:*** *The sum of these curves approximates the seasonal pattern.*

Notice that we only needed eight features (four sine / cosine pairs) to get a good estimate of the annual seasonality. Compare this to the seasonal indicator method which would have required hundreds of

Figure 6.20: Fourier features

Notice that we only needed eight features (four sine / cosine pairs) to get a good estimate of the annual seasonality. Compare this to the seasonal indicator method which would have required hundreds of features (one for each day of the year). By modeling only the "main effect" of the seasonality with Fourier features, you'll usually need to add far fewer features to your training data, which means reduced computation time and less risk of overfitting.

Choosing Fourier features with the Periodogram:

How many Fourier pairs should we actually include in our feature set? We can answer this question with the periodogram. The periodogram tells you the strength of the frequencies in a time series. Specifically, the value on the y-axis of the graph is (a ** 2 + b ** 2) / 2, where a and b are the coefficients of the sine and cosine at that frequency (as in the Fourier Components plot above).

Removing from a series its trend or seasons is called **detrending** or **deseasonalizing** the series.

Once Fourier or seasonal indicator features created, we can use the linear (nonlinear/polynomial) regression for forecasting.

6.15.3 Cycles: serial dependence (time series as features)

Some time series, have *time dependent properties*, that is, we could derive the feature directly from the time index (for example, weekly dependence). However, some other time series properties can only be modeled as serially dependent properties, that is, using as features past values of the target series. Plotted against past values (with lag) reveals the structure.

Cycles

One especially common way for serial dependence to manifest is in cycles. Cycles are patterns of growth and decay in a time series associated with how the value in a series at one time depends on values at previous times, but not necessarily on the time step itself. Cyclic behavior is characteristic of systems that can affect themselves or whose reactions persist over time. Economies, epidemics, animal populations, volcano eruptions, and similar natural phenomena often display cyclic behavior. What distinguishes cyclic behavior from seasonality is that cycles are not necessarily time dependent, as seasons are. What happens in a cycle is less about the particular date of occurrence, and more about what has happened in the recent past. The (at least relative) independence from time means that cyclic behavior can be much more irregular than seasonality.

To investigate possible serial dependence (like cycles) in a time series, we need to create "lagged" copies of the series. Lagging a time series means to

shift its values forward one or more time steps, or equivalently, to shift the times in its index backward one or more steps. In either case, the effect is that the observations in the lagged series will appear to have happened later in time. The lagged data can be used as a feature to predict the target.

Lag plot

A **lag plot** of a time series shows its values plotted against its lags. Serial dependence in a time series will often become apparent by looking at a lag plot. The most commonly used measure of serial dependence is known as **autocorrelation**, which is simply the correlation a time series has with one of its lags. Autocorrelation varies between -1 to +1.

Choosing lags

When choosing lags to use as features, it generally won't be useful to include every lag with a large autocorrelation. The partial autocorrelation tells you the correlation of a lag accounting for all of the previous lags – the amount of "new" correlation the lag contributes, so to speak. Plotting the partial autocorrelation can help you choose which lag features to use.

A plot like that above is known as a correlogram. The correlogram is for lag features essentially what the periodogram is for Fourier features.

Finally, we need to be mindful that autocorrelation and partial autocorrelation are measures of linear dependence. Because real-world time series often have substantial non-linear dependencies, it's best to look at a lag plot (or use some more general measure of dependence, like mutual information) when choosing lag features. The Sunspots series has lags with non-linear dependence which we might overlook with autocorrelation.

Lag features require that the lagged target value is known at the time being forecast. A lag 1 feature shifts the time series forward 1 step, which means you could forecast 1 step into the future but not 2 steps. we just assumed that we could always generate lags up to the period we wanted to forecast (every prediction was for just one step forward, in other words). But, in reality, we need to forecast a few steps ahead. Machine learning can help with this.

6.15.4 Forecasting with hybrid models: Components and residuals

We could imagine learning the components of a time series as an iterative process: first learn the trend and subtract it out from the series, then learn the seasonality from the detrended residuals and subtract the seasons out, then learn the cycles and subtract the cycles out, and finally only the unpredictable error remains. Add together all the components we learned and we get the complete model. This is essentially what linear regression would do if you trained it on a complete set of features modeling trend, seasons, and cycles.

We can use a single algorithm (linear regression) to learn all the components at once. But it's also possible to use one algorithm for some of the components and another algorithms for the rest. This way we can always choose the best algorithm for each component. To do this, we use one algorithm to fit the original series and then the second algorithm to fit the residual series. We'll usually want to use different feature sets (X_train_1 and X_train_2 above) depending on what we want each model to learn. If we use the first model to learn the trend, we generally wouldn't need a trend feature for the second model, for example.

1. Train and predict with first model model_1.fit(X_train_1, y_train) y_pred_1 = model_1.predict(X_train)

2. Train and predict with second model on residuals model_2.fit(X_train_2, y_train - y_pred_1) y_pred_2 = model_2.predict(X_train_2)

3. Add to get overall predictions y_pred = y_pred_1 + y_pred_2

While it's possible to use more than two models, in practice it doesn't seem to be especially helpful. In fact, the most common strategy for constructing hybrids is the one we've just described: a simple (usually linear) learning algorithm followed by a complex, non-linear learner like GBDTs or a deep neural net, the simple model typically designed as a "helper" for the powerful algorithm that follows.

There are many ways you could combine machine learning models besides the way we've outlined in this lesson. Successfully combining models, though, requires that we dig a bit deeper into how these algorithms operate.

Important remark: there are generally two ways a regression algorithm can make predictions: either by *transforming the features* or by transforming the target. Feature-transforming algorithms learn some mathematical function that takes features as an input and then combines and transforms

them to produce an output that matches the target values in the training set. Linear regression and neural nets are of this kind. *Target-transforming* algorithms use the features to group the target values in the training set and make predictions by averaging values in a group; a set of feature just indicates which group to average. Decision trees and nearest neighbors are of this kind.

Feature transformers generally can extrapolate target values beyond the training set given appropriate features as inputs, but the predictions of target transformers will always be bound within the range of the training set. If the time dummy continues counting time steps, linear regression continues drawing the trend line. Given the same time dummy, a decision tree will predict the trend indicated by the last step of the training data into the future forever. Decision trees cannot extrapolate trends. Random forests and gradient boosted decision trees (like XGBoost) are ensembles of decision trees, so they also cannot extrapolate trends.

This difference is what motivates the hybrid design in this lesson: use linear regression to extrapolate the trend, transform the target to remove the trend, and apply XGBoost to the detrended residuals. To hybridize a neural net (a feature transformer), you could instead include the predictions of another model as a feature, which the neural net would then include as part of its own predictions. The method of fitting to residuals is actually the same method the gradient boosting algorithm uses, so we will call these boosted hybrids; the method of using predictions as features is known as "stacking", so we will call these stacked hybrids.

We use linear regression to extrapolate the trend, transform the target to remove the trend, and apply XGBoost to the detrended residuals. To hybridize a neural net (a feature transformer), you could instead include the predictions of another model as a feature, which the neural net would then include as part of its own predictions. The method of fitting to residuals is actually the same method the gradient boosting algorithm uses, so we will call these **boosted hybrids**; the method of using predictions as features is known as **stacking**, so we will call these **stacked hybrids.**

6.15.5 Forecasting

There are two things to establish before designing a forecasting model:

- what information is available at the time a forecast is made (features),

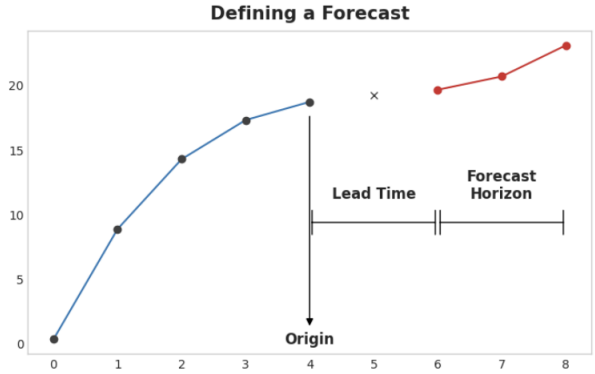
and,

- the time period during which you require forecasted values (target).

The **forecast origin** is time at which you are making a forecast. Practically, you might consider the forecast origin to be the last time for which you have training data for the time being predicted. Everything up to he origin can be used to create features.

The **forecast horizon** is the time for which you are making a forecast. We often describe a forecast by the number of time steps in its horizon: a "1-step" forecast or "5-step" forecast, say. The forecast horizon describes the target.

The time between the origin and the horizon is the **lead time** (or sometimes latency) of the forecast. A forecast's lead time is described by the number of steps from origin to horizon: a "1-step ahead" or "3-step ahead" forecast, say. In practice, it may be necessary for a forecast to begin multiple steps ahead of the origin because of delays in data acquisition or processing.



A three-step forecast horizon with a two-step lead time, using four lag features. The figure represents what would be a single row of training data -- data for a single prediction, in other words.

Figure 6.21: Defining a forecast

Prepare data for forecasting

We can create lag/lead features or time steps for linear regression. Additionally, we can create features based on the target for decision tree. How we

do this depends on the forecasting task. For example, table 6.22, shows the dataframe for 6.21 (I don't know why we need lags 5 and 6!). Note also that the lagged data is used as feature and the lead data is used as the target.

	Targets			Features				
	y_step_3	y_step_2	y_step_1	y_lag_2	y_lag_3	y_lag_4	y_lag_5	y_lag_6
Year								
2010	2	1	0	nan	nan	nan	nan	nan
2011	3	2	1	nan	nan	nan	nan	nan
2012	4	3	2	0	nan	nan	nan	nan
2013	5	4	3	1	0	nan	nan	nan
2014	6	5	4	2	1	0	nan	nan
2015	7	6	5	3	2	1	0	nan
2016	8	7	6	4	3	2	1	0
2017	9	8	7	5	4	3	2	1
2018	10	9	8	6	5	4	3	2
2019	11	10	9	7	6	5	4	3

Figure 6.22: features for a forecast

Multistep forecasting strategies

Here are strategies for producing the multiple target steps required for a forecast.

Multioutput model: Use a model that produces multiple outputs naturally. Linear regression and neural networks can both produce multiple outputs (extrapolation). This strategy is simple and efficient, but not possible for every algorithm you might want to use. XGBoost can't do this, for instance (XGBoost is only for interpolation).

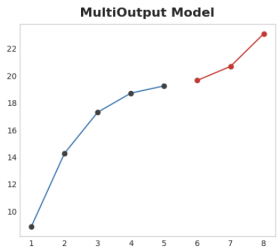


Figure 6.23: MultiOutput Model

Direct strategy Train a separate model for each step in the horizon: one

model forecasts 1-step ahead, another 2-steps ahead, and so on. Forecasting 1-step ahead is a different problem than 2-steps ahead (and so on), so it can help to have a different model make forecasts for each step. The downside is that training lots of models can be computationally expensive.

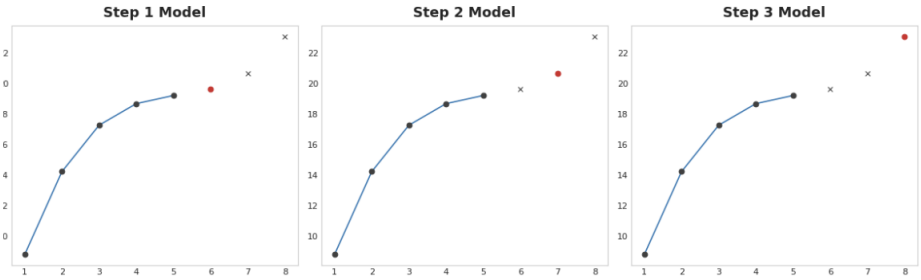


Figure 6.24: Direct strategy

Recursive strategy Train a single one-step model and use its forecasts to update the lag features for the next step. With the recursive method, we feed a model's 1-step forecast back in to that same model to use as a lag feature for the next forecasting step. We only need to train one model, but *since errors will propagate from step to step, forecasts can be inaccurate for long horizons.*

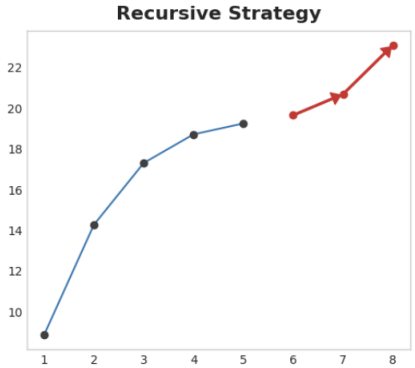


Figure 6.25: Recursive strategy

DirRec strategy A combination of the direct and recursive strategies:

train a model for each step and use forecasts from previous steps as new lag features. Step by step, each model gets an additional lag input. Since each model always has an up-to-date set of lag features, *the DirRec strategy can capture serial dependence better than Direct, but it can also suffer from error propagation like Recursive.*

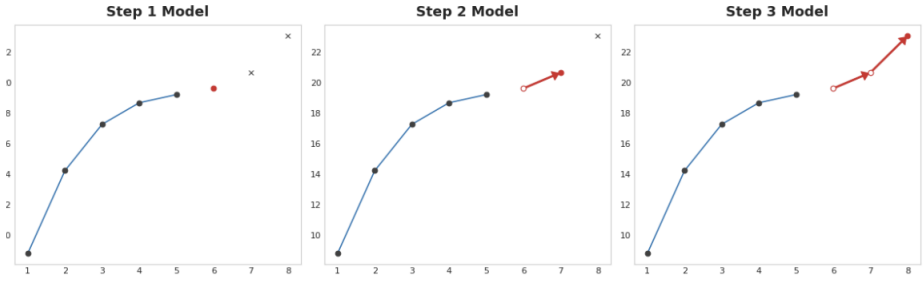


Figure 6.26: DirRec strategy

6.16 ML explainability

We want to know:

- What features in the data did the model think are most important?
- For any single prediction from a model, how did each feature in the data affect that particular prediction?
- How does each feature affects the model's predictions in a big-picture sense (what is its typical effect when considered over a large number of possible predictions)?

6.16.1 Feature importance

What features have the biggest impact on predictions? There are multiple ways to measure feature importance. Some approaches answer subtly different versions of the question above. Other approaches have documented shortcomings.

One approach is permutation importance. Compared to most other approaches, permutation importance is: fast to calculate, widely used and understood, and consistent with properties we would want a feature importance measure to have. Permutation importance is calculated after a model has been fitted. So we won't change the model or change what predictions we'd get for a given value. The idea is: If I randomly shuffle a single column of the validation data, leaving the target and all other columns in place, how would that affect the accuracy of predictions in that now-shuffled data? Randomly re-ordering a single column should cause less accurate predictions, since the resulting data no longer corresponds to anything observed in the real world. Model accuracy especially suffers if we shuffle a column that the model relied on heavily for predictions.

To use permutation with a trained model, we can PermutationImportance from eli5.sklearn.

The output for each row shows how much model performance decreased with a random shuffling (in this case, using "accuracy" as the performance metric). There is some randomness to the exact performance change from a shuffling a column. We measure the amount of randomness in our permutation importance calculation by repeating the process with multiple shuffles. The number after ± measures how performance varied from one-reshuffling to the next.

Occasionally, you may see negative values for permutation importances. In those cases, the predictions on the shuffled (or noisy) data happened to be more accurate than the real data. This happens when the feature didn't matter (should have had an importance close to 0), but random chance caused the predictions on shuffled data to be more accurate. This is more common with small datasets, because there is more room for luck/chance.

6.16.2 Partial dependence plots

While feature importance shows what variables most affect predictions, partial dependence plots show how a feature affects predictions.

This is useful to answer questions like:

- Controlling for all other house features, what impact do longitude and latitude have on home prices? To restate this, how would similarly sized houses be priced in different areas?

- Are predicted health differences between two groups due to differences in their diets, or due to some other factor?

Partial dependence plots can be interpreted similarly to the coefficients in linear or logistic regression models. Though, partial dependence plots on sophisticated models can capture more complex patterns than coefficients from simple models.

Like permutation importance, partial dependence plots are calculated after a model has been fit. The model is fit on real data that has not been artificially manipulated in any way.

To see how partial plots separate out the effect of each feature, we start by considering a single row of data. We will use the fitted model to predict our outcome. But we repeatedly alter the value for one variable (from small to large) to make a series of predictions. We trace out predicted outcomes (on the vertical axis) as we move from small values to large values (on the horizontal axis).

In this description, we used only a single row of data. Interactions between features may cause the plot for a single row to be atypical. So, we repeat that mental experiment with multiple rows from the original dataset, and we plot the average predicted outcome on the vertical axis.

2D Partial Dependence Plots: If you are curious about interactions between features, 2D partial dependence plots are also useful.

6.16.3 SHAP values

SHAP Values (an acronym from SHapley Additive exPlanations) break down a prediction to show the impact of each feature. Where could you use this?

- A model says a bank shouldn't loan someone money, and the bank is legally required to explain the basis for each loan rejection.
- A healthcare provider wants to identify what factors are driving each patient's risk of some disease so they can directly address those risk factors with targeted health interventions.

SHAP values interpret the impact of having a certain value for a given feature in comparison to the prediction we'd make if that feature took some baseline value. SHAP values do this in a way that guarantees a nice property. Specifically, you decompose a prediction with the following equation:

sum(SHAP values for all features) = pred_for_team - pred_for_baseline_values

That is, the SHAP values of all features sum up to explain why my prediction was different from the baseline. This allows us to decompose a prediction in a graph3.