ML Crash Course

Framing: Key ML Terminology

What is (supervised) machine learning? Concisely put, it is the following:

- ML systems learn how to combine input to produce useful predictions on never-before-seen data.

Let's explore fundamental machine learning terminology.

Labels

A **label** is the thing we're predicting—the `y` variable in simple linear regression. The label could be the future price of wheat, the kind of animal shown in a picture, the meaning of an audio clip, or just about anything.

Features

A **feature** is an input variable—the `x` variable in simple linear regression. A simple machine learning project might use a single feature, while a more sophisticated machine learning project could use millions of features, specified as:
x1,x2,...xN
In the spam detector example, the features could include the following:

- words in the email text
- sender's address
- time of day the email was sent
- email contains the phrase "one weird trick."

Examples

An **example** is a particular instance of data, **x.** (We put **x** in boldface to indicate that it is a vector.) We break examples into two categories:

- labeled examples
- unlabeled examples

A **labeled example** includes both feature(s) and the label. That is:

```
   labeled examples: {features, label}: (x, y)
```

Use labeled examples to **train** the model. In our spam detector example, the labeled examples would be individual emails that users have explicitly marked as "spam" or "not spam."

For example, the following table shows 5 labeled examples from a data set containing information about housing prices in California:

housingMedianAge (feature)	totalRooms (feature)	totalBedrooms (feature)	medianHouseValue (label)
15	5612	1283	66900
19	7650	1901	80100
17	720	174	85700
14	1501	337	73400
20	1454	326	65500

An **unlabeled example** contains features but not the label. That is:

```
  unlabeled examples: {features, ?}: (x, ?)
```

Here are 3 unlabeled examples from the same housing dataset, which exclude `medianHouseValue`:

housingMedianAge (feature)	totalRooms (feature)	totalBedrooms (feature)
42	1686	361
34	1226	180
33	1077	271

Once we've trained our model with labeled examples, we use that model to predict the label on unlabeled examples. In the spam detector, unlabeled examples are new emails that humans haven't yet labeled.

Models

A model defines the relationship between features and label. For example, a spam detection model might associate certain features strongly with "spam". Let's highlight two phases of a model's life:

- **Training** means creating or **learning** the model. That is, you show the model labeled examples and enable the model to gradually learn the relationships between features and label.
- **Inference** means applying the trained model to unlabeled examples. That is, you use the trained model to make useful predictions (`y'`). For example, during inference, you can predict `medianHouseValue` for new unlabeled examples.

Regression vs. classification

A **regression** model predicts continuous values. For example, regression models make predictions that answer questions like the following:

- What is the value of a house in California?
- What is the probability that a user will click on this ad?

A **classification** model predicts discrete values. For example, classification models make predictions that answer questions like the following:

- Is a given email message spam or not spam?
- Is this an image of a dog, a cat, or a hamster?

Descending into ML: Linear Regression

It has long been known that crickets (an insect species) chirp more frequently on hotter days than on cooler days. For decades, professional and amateur scientists have cataloged data on chirps-per-minute and temperature. As a birthday gift, your Aunt Ruth gives you her cricket database and asks you to learn a model to predict this relationship. Using this data, you want to explore this relationship.

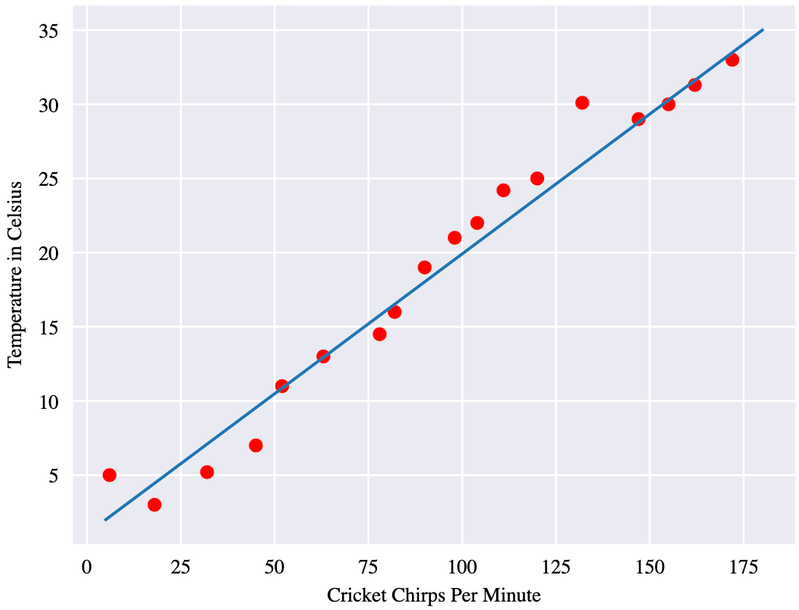


Figure 2. A linear relationship.

True, the line doesn't pass through every dot, but the line does clearly show the relationship between chirps and temperature. Using the equation for a line, you could write down this relationship as follows:

$$y = mx + b$$

where:

- y is the temperature in Celsius—the value we're trying to predict.
- m is the slope of the line.
- x is the number of chirps per minute—the value of our input feature.
- b is the y-intercept.

By convention in machine learning, you'll write the equation for a model slightly differently:

$$y' = b + w_1 x_1$$

where:

- y' is the predicted label (a desired output).
- b is the bias (the y-intercept), sometimes referred to as w_0.
- w_1 is the weight of feature 1. Weight is the same concept as the "slope" m in the traditional equation of a line.
- x_1 is a feature (a known input).

To **infer** (predict) the temperature y' for a new chirps-per-minute value x_1, just substitute the x_1 value into this model.

Although this model uses only one feature, a more sophisticated model might rely on multiple features, each having a separate weight (w_1, w_2, etc.). For example, a model that relies on three features might look as follows:

$$y' = b + w_1 x_1 + w_2 x_2 + w_3 x_3$$

Descending into ML: Training and Loss

Training a model simply means learning (determining) good values for all the weights and the bias from labeled examples. In supervised learning, a machine learning algorithm builds a model by examining many examples and attempting to find a model that minimizes loss; this process is called **empirical risk minimization**.

Loss is the penalty for a bad prediction. That is, **loss** is a number indicating how bad the model's prediction was on a single example. If the model's prediction is perfect, the loss is zero; otherwise, the loss is greater. The goal of training a model is to find a set of weights and biases that have *low* loss, on average, across all examples. For example, Figure 3 shows a high loss model on the left and a low loss model on the right. Note the following about the figure:

- The arrows represent loss.
- The blue lines represent predictions.

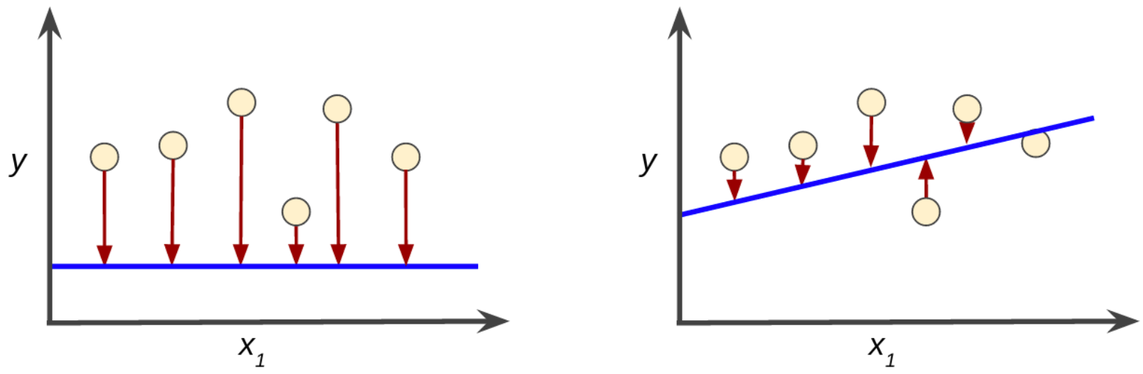


Figure 3. High loss in the left model; low loss in the right model.

Squared loss: a popular loss function

The linear regression models we'll examine here use a loss function called **squared loss** (also known as **L_2 loss**). The squared loss for a single example is as follows:

```
= the square of the difference between the label and the prediction
= (observation - prediction(x))^2
= (y - y')^2
```

Mean square error (**MSE**) is the average squared loss per example over the whole dataset. To calculate MSE, sum up all the squared losses for individual examples and then divide by the number of examples:

$$MSE = \frac{1}{N} \sum_{(x,y) \in D} (y - prediction(x))^2$$

where:

- (x, y) is an example in which
  - x is the set of features (for example, chirps/minute, age, gender) that the model uses to make predictions.
  - y is the example's label (for example, temperature).
- $prediction(x)$ is a function of the weights and bias in combination with the set of features x.
- D is a data set containing many labeled examples, which are (x, y) pairs.
- N is the number of examples in D.

Although MSE is commonly-used in machine learning, it is neither the only practical loss function nor the best loss function for all circumstances.

Descending into ML: Reducing Loss

- **Iterative Approach**

A Machine Learning model is trained by starting with an initial guess for the weights and bias and iteratively adjusting those guesses until learning the weights and bias with the lowest possible loss.

- **Gradient Descent**

A technique to minimize loss by computing the gradients of loss with respect to the model's parameters, conditioned on training data. Informally, gradient descent iteratively adjusts parameters, gradually finding the best combination of weights and bias to minimize loss.

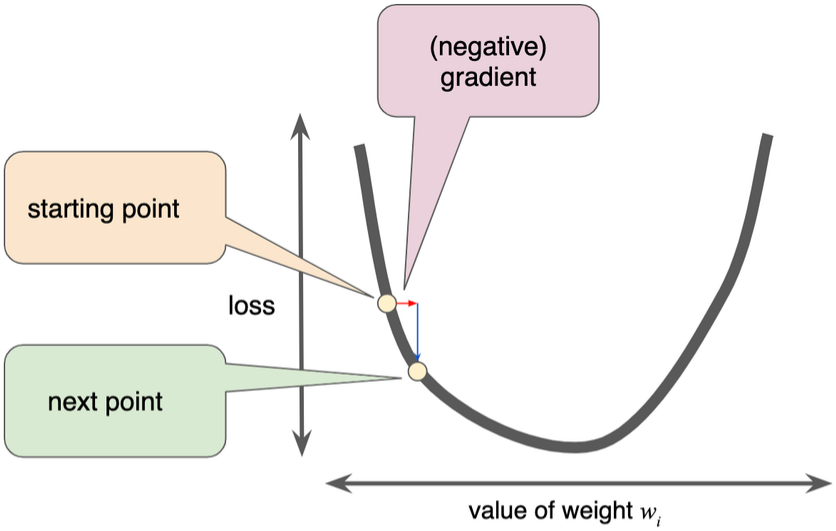


Figure 5. A gradient step moves us to the next point on the loss curve.

The gradient descent then repeats this process, edging ever closer to the minimum.

★ **Note:** When performing gradient descent, we generalize the above process to tune all the model parameters *simultaneously*. For example, to find the optimal values of both w_1 and the bias b, we calculate the gradients with respect to both w_1 and b. Next, we modify the values of w_1 and b based on their respective gradients. Then we repeat these steps until we reach minimum loss.

For convex shaped problems

- **Learning Rate**

A scalar used to train a model via gradient descent. During each iteration, the **gradient descent** algorithm multiplies the learning rate by the gradient. The resulting product is called the **gradient step**. Learning rate is a key **hyperparameter**.

- **Stochastic Gradient Descent (SGD)**

A **gradient descent** algorithm in which the batch size is one. In other words, SGD relies on a single example chosen uniformly at random from a dataset to calculate an estimate of the gradient at each step.

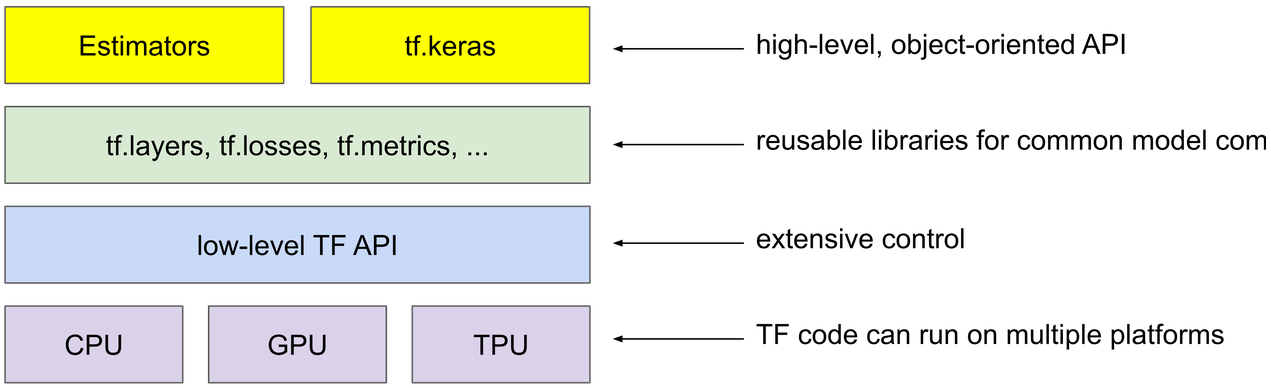
- **Mini Batch SGD**

A **gradient descent** algorithm that uses **mini-batches**. In other words, mini-batch stochastic gradient descent estimates the gradient based on a small subset of the training data.

Introduction to TensorFlow

- TensorFlow is an end-to-end open source platform for machine learning. TensorFlow is a rich system for managing all aspects of a machine learning system; however, this class focuses on using a particular TensorFlow API to develop and train machine learning models. See the TensorFlow documentation for complete details on the broader TensorFlow system.
- TensorFlow APIs are arranged hierarchically, with the high-level APIs built on the low-level APIs. Machine learning researchers use the low-level APIs to create and explore new machine learning algorithms.

The following figure shows the hierarchy of TensorFlow toolkits:



TensorFlow toolkit hierarchy

Summary of hyperparameter tuning

Most machine learning problems require a lot of hyperparameter tuning. Unfortunately, we can't provide concrete tuning rules for every model. Lowering the learning rate can help one model converge efficiently but make another model converge much too slowly. You must experiment to find the best set of hyperparameters for your dataset. That said, here are a few rules of thumb:

- Training loss should steadily decrease, steeply at first, and then more slowly until the slope of the curve reaches or approaches zero.
- If the training loss does not converge, train for more epochs.
- If the training loss decreases too slowly, increase the learning rate. Note that setting the learning rate too high may also prevent training loss from converging.
- If the training loss varies wildly (that is, the training loss jumps around), decrease the learning rate.
- Lowering the learning rate while increasing the number of epochs or the batch size is often a good combination.

- Setting the batch size to a *very* small batch number can also cause instability. First, try large batch size values. Then, decrease the batch size until you see degradation.
- For real-world datasets consisting of a very large number of examples, the entire dataset might not fit into memory. In such cases, you'll need to reduce the batch size to enable a batch to fit into memory.

Remember: the ideal combination of hyperparameters is data dependent, so you must always experiment and verify.

Generalization: Peril of Overfitting

Overfitting occurs when a model tries to fit the training data so closely that it does not generalize well to new data.

A machine learning model aims to make good predictions on new, previously unseen data. But if you are building a model from your data set, how would you get the previously unseen data? Well, one way is to divide your data set into two subsets:

- **training set**—a subset to train a model.
- **test set**—a subset to test the model.

Good performance on the test set is a useful indicator of good performance on the new data in general, assuming that:

- The test set is large enough.
- You don't cheat by using the same test set over and over.

Figure 2. Slicing a single data set into three subsets.

Use the **validation set** to evaluate results from the training set. Then, use the test set to double-check your evaluation *after* the model has "passed" the validation set. The following figure shows this new workflow:

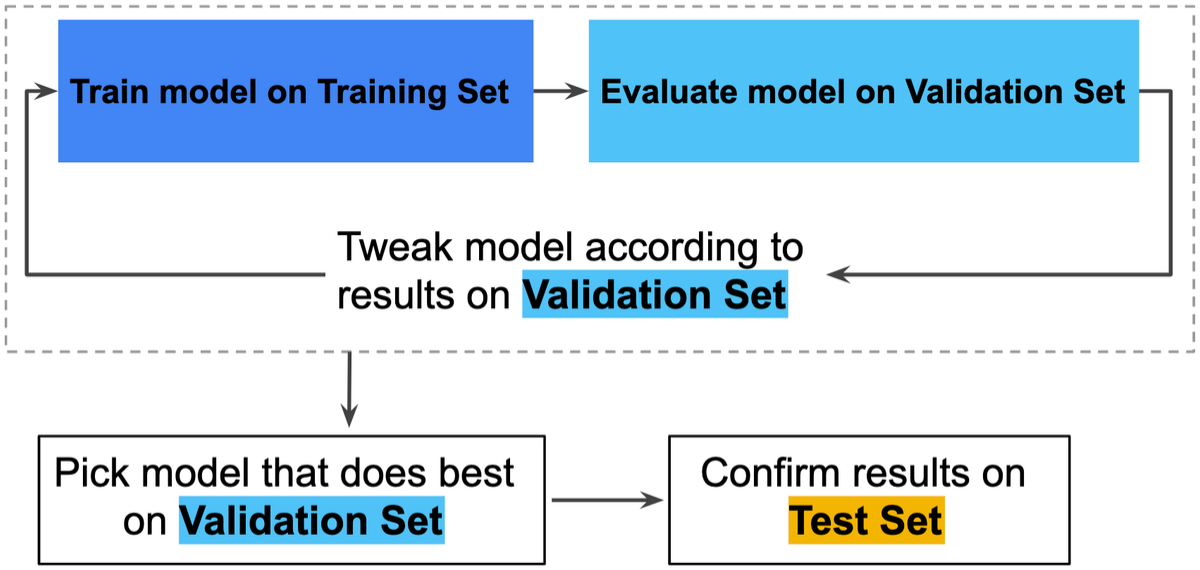


Figure 3. A better workflow.

Test sets and validation sets "wear out" with repeated use. That is, the more you use the same data to make decisions about hyperparameter settings or other model improvements, the less confidence you'll have that these results actually generalize to new, unseen data.
If possible, it's a good idea to collect more data to "refresh" the test set and validation set. Starting anew is a great reset.

Representation: Feature Engineering

- The process of determining which **features** might be useful in training a model, and then converting raw data from log files and other sources into said features.
- In TensorFlow, feature engineering often means converting raw log file entries to **tf.Example** protocol buffers.

- Feature engineering is sometimes called **feature extraction**.
- Feature engineering means transforming raw data into a feature vector. Expect to spend significant time doing feature engineering.
- Many machine learning models must represent the features as real-numbered vectors since the feature values must be multiplied by the model weights.

One-hot encoding for feature transformation
A sparse vector in which:
- One element is set to 1.
- All other elements are set to 0.

One-hot encoding is commonly used to represent strings or identifiers that have a finite set of possible values. For example, suppose a given botany dataset chronicles 15,000 different species, each denoted with a unique string identifier. As part of feature engineering, you'll probably encode those string identifiers as one-hot vectors in which the vector has a size of 15,000.

Avoid rarely used discrete feature values
Good feature values should appear more than 5 or so times in a data set. Doing so enables a model to learn how this feature value relates to the label. Conversely, if a feature's value appears only once or very rarely, the model can't make predictions based on that feature.

Prefer clear and obvious meanings
Each feature should have a clear and obvious meaning to anyone on the project. For example, the following good feature is clearly named and the value makes sense with respect to the name: `house_age_years: 27`

Don't mix "magic" values with actual data
Good floating-point features don't contain peculiar out-of-range discontinuities or "magic" values. For example, suppose a feature holds a floating-point value between 0 and 1 ex. `quality_rating: 0.82` However, if a user didn't enter a `quality_rating`, perhaps the data set represented its absence with a magic value like the following: `quality_rating: -1`

Account for upstream instability
The definition of a feature shouldn't change over time. For example, the following value is useful because the city name *probably* won't change.

Scaling feature values

Scaling means converting floating-point feature values from their natural range (for example, 100 to 900) into a standard range (for example, 0 to 1 or -1 to +1). If a feature set consists of only a single feature, then scaling provides little to no practical benefit. If, however, a feature set consists of multiple features, then feature scaling provides the following benefits:

- Helps gradient descent converge more quickly.
- Helps avoid the "NaN trap," in which one number in the model becomes a NaN (e.g., when a value exceeds the floating-point precision limit during training), and —due to math operations—every other number in the model also eventually becomes a NaN.
- Helps the model learn appropriate weights for each feature. Without feature scaling, the model will pay too much attention to the features having a wider range.

You don't have to give every floating-point feature exactly the same scale. Nothing terrible will happen if Feature A is scaled from -1 to +1 while Feature B is scaled from -3 to +3. However, your model will react poorly if Feature B is scaled from 5000 to 100000.

Handling extreme outliers

Maybe cap extreme feature values

Binning

Converting a (usually **continuous**) feature into multiple binary features called buckets or bins, typically based on value range. For example, instead of representing temperature as a single continuous floating-point feature, you could chop ranges of temperatures into discrete bins. Given temperature data sensitive to a tenth of a degree, all temperatures between 0.0 and 15.0 degrees could be put into one bin, 15.1 to 30.0 degrees could be a second bin, and 30.1 to 50.0 degrees could be a third bin.

Scrubbing

Until now, we've assumed that all the data used for training and testing was trustworthy. In real-life, many examples in data sets are unreliable due to one or more of the following:

- **Omitted values.** For instance, a person forgot to enter a value for a house's age.

- **Duplicate examples.** For example, a server mistakenly uploaded the same logs twice.
- **Bad labels.** For instance, a person mislabeled a picture of an oak tree as a maple.
- **Bad feature values.** For example, someone typed in an extra digit, or a thermometer was left out in the sun.

Once detected, you typically "fix" bad examples by removing them from the data set. To detect omitted values or duplicated examples, you can write a simple program. Detecting bad feature values or labels can be far trickier.

Know your data

Follow these rules:

- Keep in mind what you think your data should look like.
- Verify that the data meets these expectations (or that you can explain why it doesn't).
- Double-check that the training data agrees with other sources (for example, dashboards).

Treat your data with all the care that you would treat any mission-critical code. Good ML relies on good data.

Feature Crosses

- A **feature cross** is a **synthetic feature** formed by multiplying (crossing) two or more features. Crossing combinations of features can provide predictive abilities beyond what those features can provide individually. A feature cross is formed by crossing (taking a Cartesian product of) individual binary features obtained from **categorical data** or from **continuous features** via **bucketing**.
- Feature crosses help represent nonlinear relationships.

ex. Housing Price Predictor
[latitude x No. of rooms] is a good feature cross as number of rooms in SF is not the same as number of rooms in Bakersfield

Kinds of feature crosses

We can create many different kinds of feature crosses. For example:

- `[A X B]`: a feature cross formed by multiplying the values of two features.

- `[A x B x C x D x E]`: a feature cross formed by multiplying the values of five features.
- `[A x A]`: a feature cross formed by squaring a single feature.

Thanks to stochastic gradient descent, linear models can be trained efficiently. Consequently, supplementing scaled linear models with feature crosses has traditionally been an efficient way to train on massive-scale data sets.

Crossing One-Hot Vectors

Think of feature crosses of one-hot feature vectors as logical conjunctions. For example, suppose we have two features: country and language. A one-hot encoding of each generates vectors with binary features that can be interpreted as `country=USA, country=France` or `language=English, language=Spanish`. Then, if you do a feature cross of these one-hot encodings, you get binary features that can be interpreted as logical conjunctions, such as:

```
country:usa AND language:spanish
```

This gives us a vastly more predictive ability than either feature on its own.

Linear learners scale well to massive data. Using feature crosses on massive data sets is one efficient strategy for learning highly complex models. Neural networks provide another strategy.

Regularization for Simplicity

The penalty on a model's complexity. Regularization helps prevent **overfitting**. Different kinds of regularization include:

- **L_1 regularization**
- **L_2 regularization**
- **dropout regularization**
- **early stopping** (this is not a formal regularization method, but can effectively limit overfitting)

Consider the following **generalization curve**, which shows the loss for both the training set and validation set against the number of training iterations.

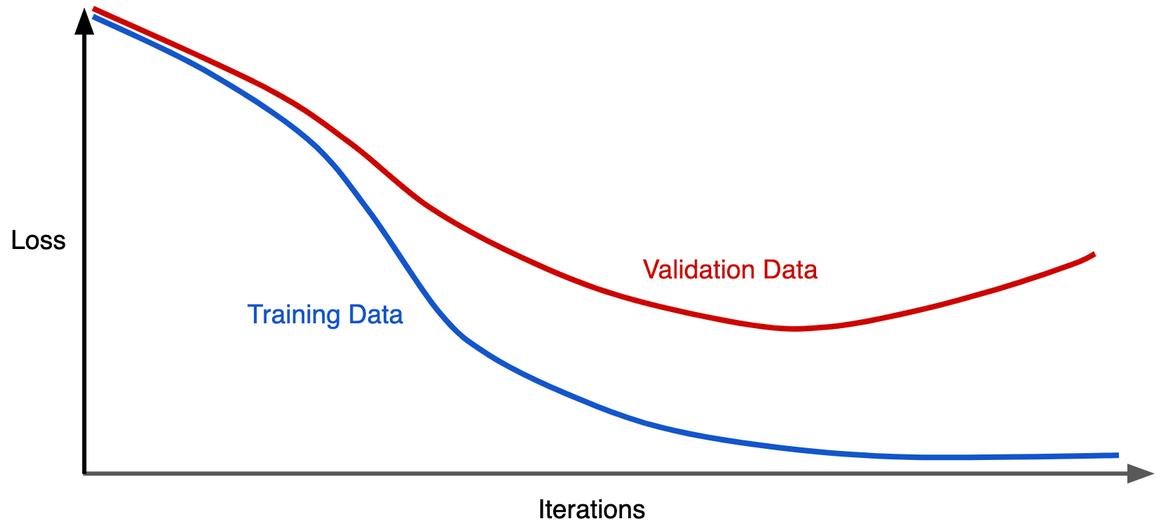


Figure 1. Loss on training set and validation set.

Figure 1 shows a model in which training loss gradually decreases, but validation loss eventually goes up. In other words, this generalization curve shows that the model is overfitting to the data in the training set. Channeling our inner Ockham, perhaps we could prevent overfitting by penalizing complex models, a principle called **regularization**.

In other words, instead of simply aiming to minimize loss (empirical risk minimization):

$$\text{minimize}(\text{Loss}(\text{Data}|\text{Model}))$$

we'll now minimize loss+complexity, which is called **structural risk minimization**:

$$\text{minimize}(\text{Loss}(\text{Data}|\text{Model}) + \text{complexity}(\text{Model}))$$

Our training optimization algorithm is now a function of two terms: the **loss term**, which measures how well the model fits the data, and the **regularization term**, which measures model complexity.

Machine Learning Crash Course focuses on two common (and somewhat related) ways to think of model complexity:

- Model complexity as a function of the *weights* of all the features in the model.
- Model complexity as a function of the *total number of features* with nonzero weights. (A later module covers this approach.)

If model complexity is a function of weights, a feature weight with a high absolute value is more complex than a feature weight with a low absolute value.

We can quantify complexity using the ***L_2* regularization** formula, which defines the regularization term as the sum of the squares of all the feature weights:

$$L_2 \text{ regularization term} = ||\boldsymbol{w}||_2^2 = w_1^2 + w_2^2 + \ldots + w_n^2$$

regularization rate

A scalar value, represented as lambda, specifying the relative importance of the regularization function. The following simplified **loss** equation shows the regularization rate's influence:

$$\text{minimize}(\text{loss function} + \lambda(\text{regularization function}))$$

Raising the regularization rate reduces **overfitting** but may make the model less **accurate**.

Regularization Rate also called as lambda

Logistic Regression

- Many problems require a probability estimate as output. Logistic regression is an extremely efficient mechanism for calculating probabilities. Practically speaking, you can use the returned probability in either of the following two ways:
  - "As is"
  - Converted to a binary category.
- Instead of predicting *exactly* 0 or 1, **logistic regression** generates a probability—a value *between* 0 and 1, exclusive. For example, consider a logistic regression model for spam detection. If the model infers a value of 0.932 on a particular email message, it implies a 93.2% probability that the email message is spam.

Loss function for Logistic Regression

The loss function for linear regression is squared loss. The loss function for logistic regression is **Log Loss**, which is defined as follows:

$$\text{Log Loss} = \sum_{(x,y)\in D} -y\log(y') - (1-y)\log(1-y')$$

where:

- $(x, y) \in D$ is the data set containing many labeled examples, which are (x, y) pairs.
- y is the label in a labeled example. Since this is logistic regression, every value of y must either be 0 or 1.
- y' is the predicted value (somewhere between 0 and 1), given the set of features in x.

Regularization in Logistic Regression

Regularization is extremely important in logistic regression modeling. Without regularization, the asymptotic nature of logistic regression would keep driving loss towards 0 in high dimensions. Consequently, most logistic regression models use one of the following two strategies to dampen model complexity:

- L_2 regularization.
- Early stopping, that is, limiting the number of training steps or the learning rate.

(We'll discuss a third strategy—L_1 regularization—in a later module.)

Imagine that you assign a unique id to each example, and map each id to its own feature. If you don't specify a regularization function, the model will become completely overfit. That's because the model would try to drive loss to zero on all examples and never get there, driving the weights for each indicator feature to +infinity or -infinity. This can happen in high dimensional data with feature crosses, when there's a huge mass of rare crosses that happen only on one example each.

Fortunately, using L_2 or early stopping will prevent this problem.

Classification

Logistic regression returns a probability. You can use the returned probability "as is" (for example, the probability that the user will click on this ad is 0.00023) or convert the returned probability to a binary value (for example, this email is spam).

A logistic regression model that returns 0.9995 for a particular email message is predicting that it is very likely to be spam. Conversely, another email message with a prediction score of 0.0003 on that same logistic regression model is very likely not spam. However, what about an email message with a prediction score of 0.6? In order to map a logistic regression value to a binary category, you must define a **classification threshold** (also called the **decision threshold**). A value above that threshold indicates "spam"; a value below indicates "not spam." It is tempting to assume that the classification threshold should always be 0.5, but thresholds are problem-dependent, and are therefore values that you must tune.

The following sections take a closer look at metrics you can use to evaluate a classification model's predictions, as well as the impact of changing the classification threshold on these predictions.

★ **Note:** "Tuning" a threshold for logistic regression is different from tuning hyperparameters such as learning rate. Part of choosing a threshold is assessing how much you'll suffer for making a mistake. For example, mistakenly labeling a non-spam message as spam is very bad. However, mistakenly labeling a spam message as non-spam is unpleasant, but hardly the end of your job.

Accuracy is one metric for evaluating classification models. Informally, **accuracy** is the fraction of predictions our model got right. Formally, accuracy has the following definition:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

positive class

In **binary classification**, the two possible classes are labeled as positive and negative. The positive outcome is the thing we're testing for. (Admittedly, we're simultaneously testing for both outcomes, but play along.) For example, the positive class in a medical test might be "tumor." The positive class in an email classifier might be "spam."

Contrast with **negative class**.

precision

A metric for **classification models**. Precision identifies the frequency with which a model was correct when predicting the **positive class**. That is:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

recall

A metric for **classification models** that answers the following question: Out of all the possible positive labels, how many did the model correctly identify? That is:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Consider two models—A and B—that each evaluate the same dataset. Which one of the following statements is true?

If model A has better recall than model B, then model A is better.

If model A has better precision and better recall than model B, then model A is probably better. ✓

In general, a model that outperforms another model on both precision and recall is likely the better model. Obviously, we'll need to make sure that comparison is being done at a precision / recall point that is useful in practice for this to be meaningful. For example, suppose our spam detection model needs to have at least 90% precision to be useful and avoid unnecessary false alarms. In this case, comparing one model at {20% precision, 99% recall} to another at {15% precision, 98% recall} is not particularly instructive, as neither model meets the 90% precision requirement. But with that caveat in mind, this is a good way to think about comparing models when using precision and recall.

Correct answer.

prediction bias

A value indicating how far apart the average of **predictions** is from the average of **labels** in the dataset.

Not to be confused with the **bias term** in machine learning models or with **bias in ethics and fairness**.

L_1 regularization

A type of **regularization** that penalizes weights in proportion to the sum of the absolute values of the weights. In models relying on **sparse features**, L_1 regularization helps drive the weights of irrelevant or barely relevant features to exactly 0, which removes those features from the model. Contrast with **L_2 regularization**.

Imagine a linear model with 100 input features:

- 10 are highly informative.
- 90 are non-informative.

Assume that all features have values between -1 and 1. Which of the following statements are true?

L1 regularization may cause informative features to get a weight of exactly 0.0.

L1 regularization will encourage most of the non-informative weights to be exactly 0.0. ✓

L1 regularization of sufficient lambda tends to encourage non-informative weights to become exactly 0.0. By doing so, these non-informative features leave the model.

1 of 2 correct answers.

L1 regularization will encourage many of the non-informative weights to be nearly (but not exactly) 0.0.

L_1 vs. L_2 Regularization

Explore the options below.

Imagine a linear model with 100 input features, all having values between -1 and 1:

- 10 are highly informative.
- 90 are non-informative.

Which type of regularization will produce the smaller model?

L_2 regularization.

L_1 regularization. ✓

L_1 regularization tends to reduce the number of features. In other words, L_1 regularization often reduces the model size.

Correct answer.

Neural Networks

If you recall from the Feature Crosses unit, the following classification problem is nonlinear:

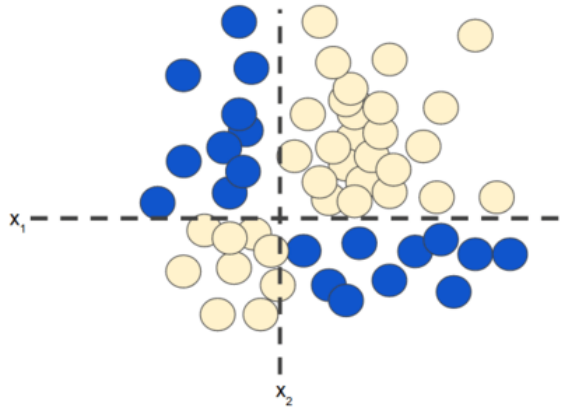


Figure 1. Nonlinear classification problem.

"Nonlinear" means that you can't accurately predict a label with a model of the form $b + w_1 x_1 + w_2 x_2$ In other words, the "decision surface" is not a line. Previously, we looked at feature crosses as one possible approach to modeling nonlinear problems.

Now consider the following data set:

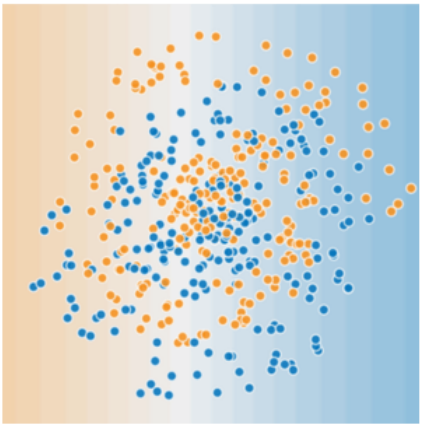


Figure 2. A more difficult nonlinear classification problem.

The data set shown in Figure 2 can't be solved with a linear model.

Non Linear Problem

To see how neural networks might help with nonlinear problems, let's start by representing a linear model as a graph:

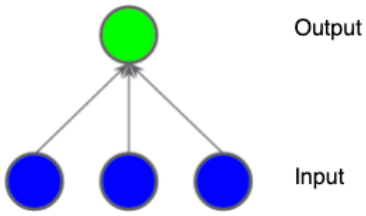


Figure 3. Linear model as graph.

Each blue circle represents an input feature, and the green circle represents the weighted sum of the inputs.

How can we alter this model to improve its ability to deal with nonlinear problems?

Linear Model

Hidden Layers

In the model represented by the following graph, we've added a "hidden layer" of intermediary values. Each yellow node in the hidden layer is a weighted sum of the blue input node values. The output is a weighted sum of the yellow nodes.

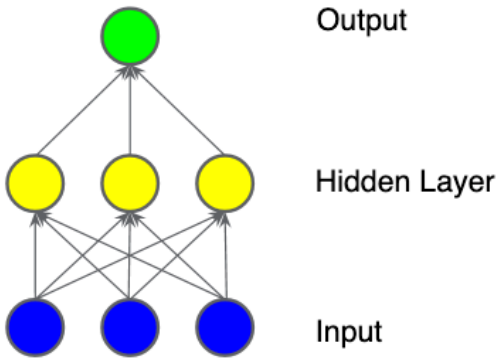


Figure 4. Graph of two-layer model.

Is this model linear? Yes—its output is still a linear combination of its inputs.

In the model represented by the following graph, we've added a second hidden layer of weighted sums.

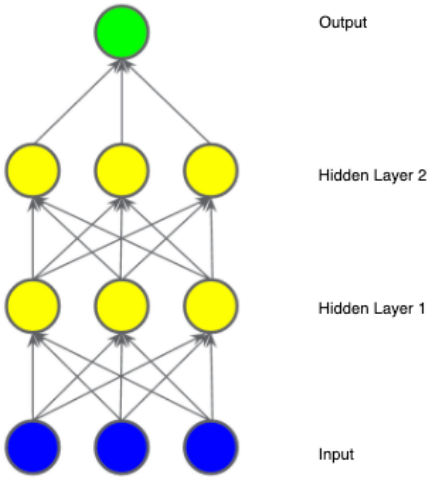


Figure 5. Graph of three-layer model.

Is this model still linear? Yes, it is. When you express the output as a function of the input and simplify, you get just another weighted sum of the inputs. This sum won't effectively model the nonlinear problem in Figure 2.

Linear Model with Hidden Layers

Activation Functions

To model a nonlinear problem, we can directly introduce a nonlinearity. We can pipe each hidden layer node through a nonlinear function.

In the model represented by the following graph, the value of each node in Hidden Layer 1 is transformed by a nonlinear function before being passed on to the weighted sums of the next layer. This nonlinear function is called the activation function.

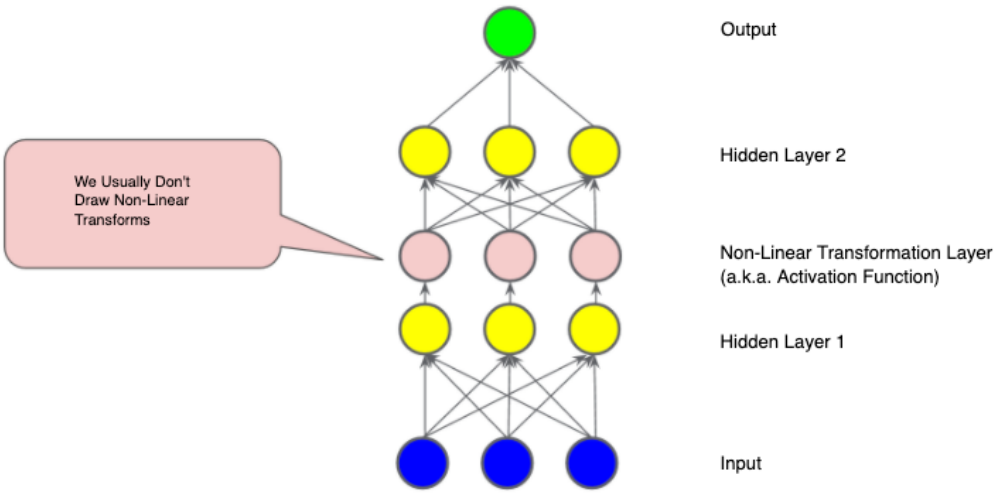


Figure 6. Graph of three-layer model with activation function.

Now that we've added an activation function, adding layers has more impact. Stacking nonlinearities on nonlinearities lets us model very complicated relationships between the inputs and the predicted outputs. In brief, each layer is effectively learning a more complex, higher-level function over the raw inputs. If you'd like to develop more intuition on how this works, see Chris Olah's excellent blog post.

Non Linear Model

Common Activation Functions

The following **sigmoid** activation function converts the weighted sum to a value between 0 and 1.

$$F(x) = \frac{1}{1 + e^{-x}}$$

Here's a plot:

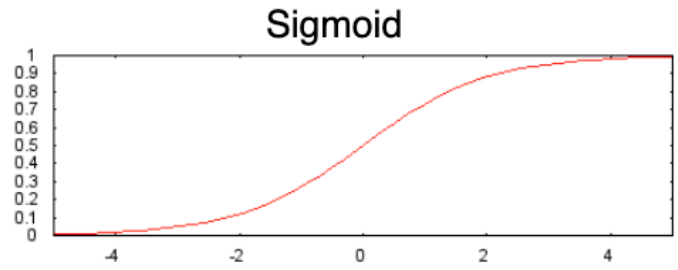


Figure 7. Sigmoid activation function.

The following **rectified linear unit** activation function (or **ReLU**, for short) often works a little better than a smooth function like the sigmoid, while also being significantly easier to compute.

$$F(x) = max(0, x)$$

The superiority of ReLU is based on empirical findings, probably driven by ReLU having a more useful range of responsiveness. A sigmoid's responsiveness falls off relatively quickly on both sides.

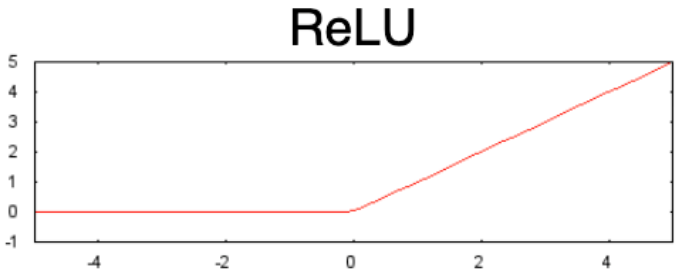
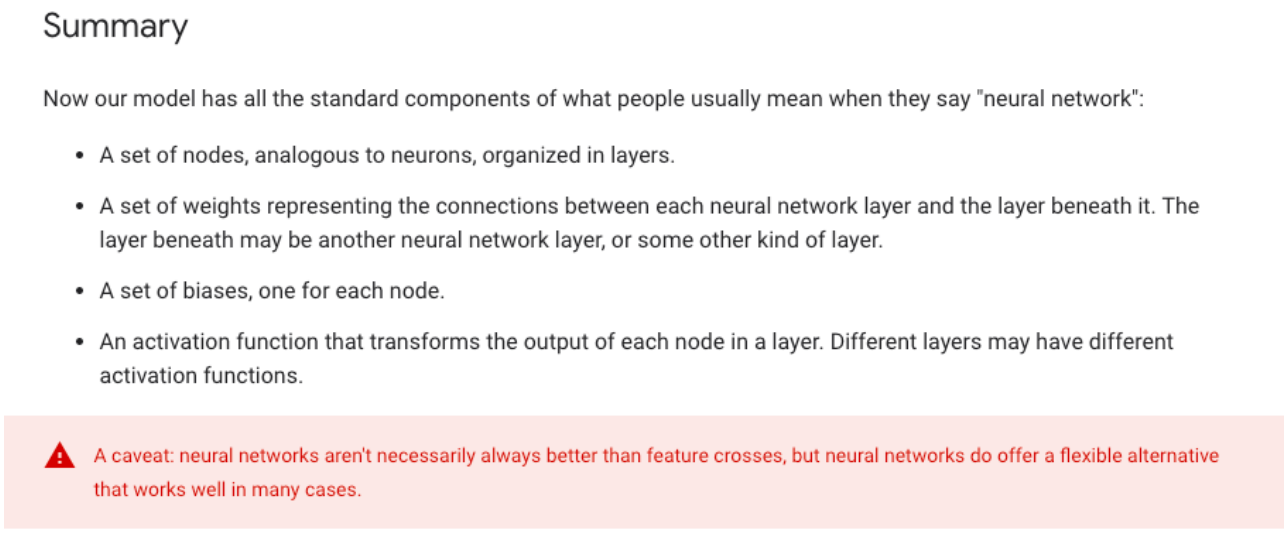


Figure 8. ReLU activation function.

In fact, any mathematical function can serve as an activation function. Suppose that σ represents our activation function (Relu, Sigmoid, or whatever). Consequently, the value of a node in the network is given by the following formula:

$$\sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b)$$

TensorFlow provides out-of-the-box support for many activation functions. You can find these activation functions within TensorFlow's list of wrappers for primitive neural network operations. That said, we still recommend starting with ReLU.



neural network

A model that, taking inspiration from the brain, is composed of layers (at least one of which is **hidden**) consisting of simple connected units or **neurons** followed by nonlinearities.

neuron

A node in a **neural network**, typically taking in multiple input values and generating one output value. The neuron calculates the output value by applying an **activation function** (nonlinear transformation) to a weighted sum of input values.

Multi-Class Neural Networks: One vs. All

One vs. all provides a way to leverage binary classification. Given a classification problem with N possible solutions, a one-vs.-all solution consists of N separate binary classifiers—one binary classifier for each possible outcome. During training, the model runs through a sequence of binary classifiers, training each to answer a separate classification question. For example, given a picture of a dog, five different recognizers might be trained, four seeing the image as a negative example (not a dog) and one seeing the image as a positive example (a dog). That is:

1. Is this image an apple? No.
2. Is this image a bear? No.
3. Is this image candy? No.
4. Is this image a dog? Yes.
5. Is this image an egg? No.

This approach is fairly reasonable when the total number of classes is small, but becomes increasingly inefficient as the number of classes rises.

We can create a significantly more efficient one-vs.-all model with a deep neural network in which each output node represents a different class. The following figure suggests this approach:

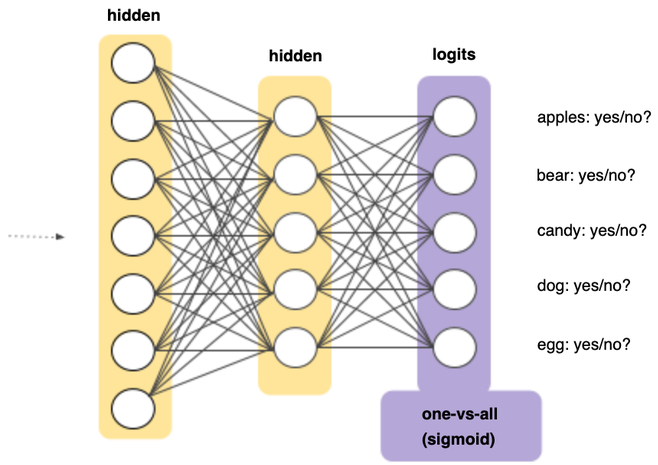


Figure 1. A one-vs.-all neural network.

Recall that logistic regression produces a decimal between 0 and 1.0. For example, a logistic regression output of 0.8 from an email classifier suggests an 80% chance of an email being spam and a 20% chance of it being not spam. Clearly, the sum of the probabilities of an email being either spam or not spam is 1.0.

Softmax extends this idea into a multi-class world. That is, Softmax assigns decimal probabilities to each class in a multi-class problem. Those decimal probabilities must add up to 1.0. This additional constraint helps training converge more quickly than it otherwise would.

For example, returning to the image analysis we saw in Figure 1, Softmax might produce the following likelihoods of an image belonging to a particular class:

Class	Probability
apple	0.001
bear	0.04
candy	0.008
dog	0.95
egg	0.001

Softmax is implemented through a neural network layer just before the output layer. The Softmax layer must have the same number of nodes as the output layer.

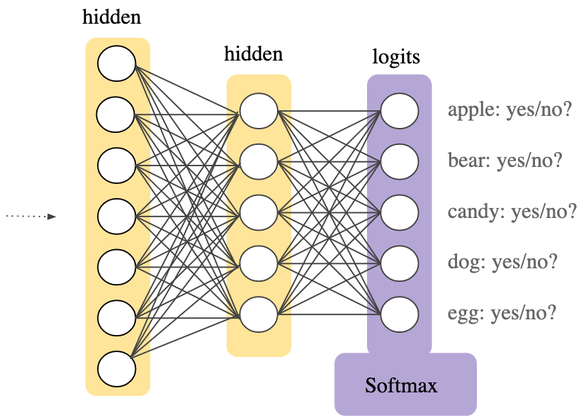


Figure 2. A Softmax layer within a neural network.

Softmax assumes that each example is a member of exactly one class. Some examples, however, can simultaneously be a member of multiple classes. For such examples:

- You may not use Softmax.
- You must rely on multiple logistic regressions.

For example, suppose your examples are images containing exactly one item—a piece of fruit. Softmax can determine the likelihood of that one item being a pear, an orange, an apple, and so on. If your examples are images containing all sorts of things—bowls of different kinds of fruit—then you'll have to use multiple logistic regressions instead.

Embeddings

- An **embedding** is a relatively low-dimensional space into which you can translate high-dimensional vectors. Embeddings make it easier to do machine learning on large inputs like sparse vectors representing words.
- Ideally, an embedding captures some of the semantics of the input by placing semantically similar inputs close together in the embedding space. An embedding can be learned and reused across models.
- **Collaborative filtering** is the task of making predictions about the interests of a user based on interests of many other users. As an example, let's look at the task of movie recommendation. Suppose we have 1,000,000 users, and a list of the movies each user has watched (from a catalog of 500,000 movies). Our goal is to recommend movies to users.
- To solve this problem some method is needed to determine which movies are similar to each other. We can achieve this goal by embedding the movies into a low-dimensional space created such that similar movies are nearby.
- **Categorical data** refers to input features that represent one or more discrete items from a finite set of choices. For example, it can be the set of movies a user has watched, the set of words in a document, or the occupation of a person.
- Categorical data is most efficiently represented via **sparse tensors**, which are tensors with very few non-zero elements. For example, if we're building a movie recommendation model, we can assign a unique ID to each possible movie, and then represent each user by a sparse tensor of the movies they have watched, as shown in Figure 3.

A sample input for our movie recommendation problem.

- Each row of the matrix in above figure is an example capturing a user's movie-viewing history, and is represented as a sparse tensor because each user only watches a small fraction of all possible movies. The last row corresponds to the sparse tensor [1, 3, 999999], using the vocabulary indices shown above the movie icons.
- Likewise one can represent words, sentences, and documents as sparse vectors where each word in the vocabulary plays a role similar to the movies in our recommendation example.
- In order to use such representations within a machine learning system, we need a way to represent each sparse vector as a vector of numbers so that semantically similar items (movies or words) have similar distances in the vector space. But how do you represent a word as a vector of numbers?
- The simplest way is to define a giant input layer with a node for every word in your vocabulary, or at least a node for every word that appears in your data. If 500,000 unique words appear in your data, you could represent a word with a length 500,000 vector and assign each word to a slot in the vector.
- If you assign "horse" to index 1247, then to feed "horse" into your network you might copy a 1 into the 1247th input node and 0s into all the rest. This sort of representation is called a **one-hot encoding**, because only one index has a non-zero value.
- Huge input vectors mean a super-huge number of weights for a neural network. If there are M words in your vocabulary and N nodes in the first layer of the network above the input, you have MxN weights to train for that layer. A large number of weights causes further problems:
  - **Amount of data**. The more weights in your model, the more data you need to train effectively.
  - **Amount of computation**. The more weights, the more computation required to train and use the model. It's easy to exceed the capabilities of your hardware.

Lack of Meaningful Relations Between Vectors

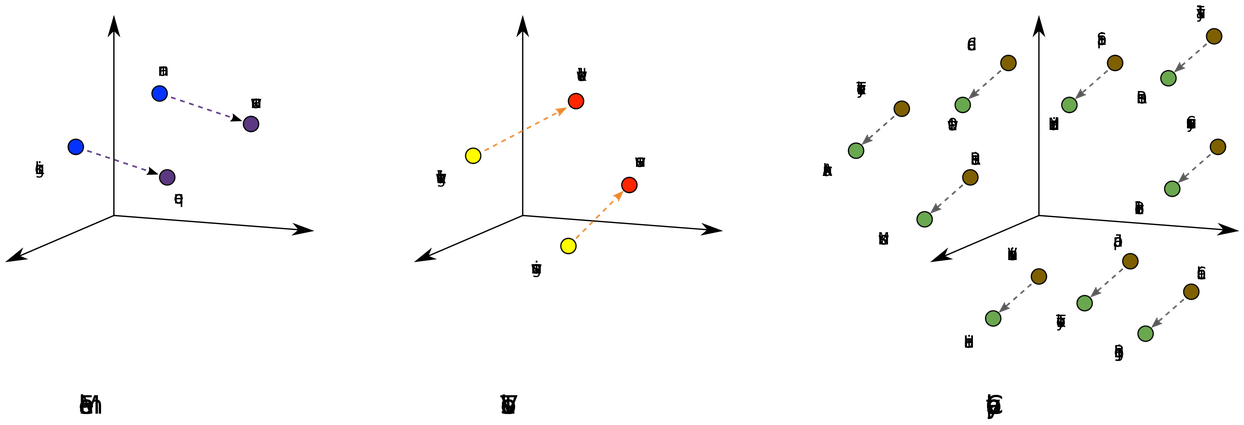
If you feed the pixel values of RGB channels into an image classifier, it makes sense to talk about "close" values. Reddish blue is close to pure blue, both semantically and in terms of the geometric distance between vectors. But a vector with a 1 at index 1247 for "horse" is not any closer to a vector with a 1 at index 50,430 for "antelope" than it is to a vector with a 1 at index 238 for "television".

The Solution: Embeddings

The solution to these problems is to use **embeddings**, which translate large sparse vectors into a lower-dimensional space that preserves semantic relationships. We'll explore embeddings intuitively, conceptually, and programmatically in the following sections of this module.

Embeddings: Translating to a Lower-Dimensional Space

You can solve the core problems of sparse input data by mapping your high-dimensional data into a lower-dimensional space.
As you can see from the paper exercises, even a small multi-dimensional space provides the freedom to group semantically similar items together and keep dissimilar items far apart. Position (distance and direction) in the vector space can encode semantics in a good embedding. For example, the following visualizations of real embeddings show geometrical relationships that capture semantic relations like the relation between a country and its capital:



Three examples of word embeddings that represent word relationships geometrically: gender (man/woman and king/queen), verb tense (walking/walked and swimming/swam), and capital cities (Turkey/Ankara and Vietnam/Hanoi)

Figure: Embeddings can produce remarkable analogies.
This sort of meaningful space gives your machine learning system opportunities to detect patterns that may help with the learning task.

Shrinking the network

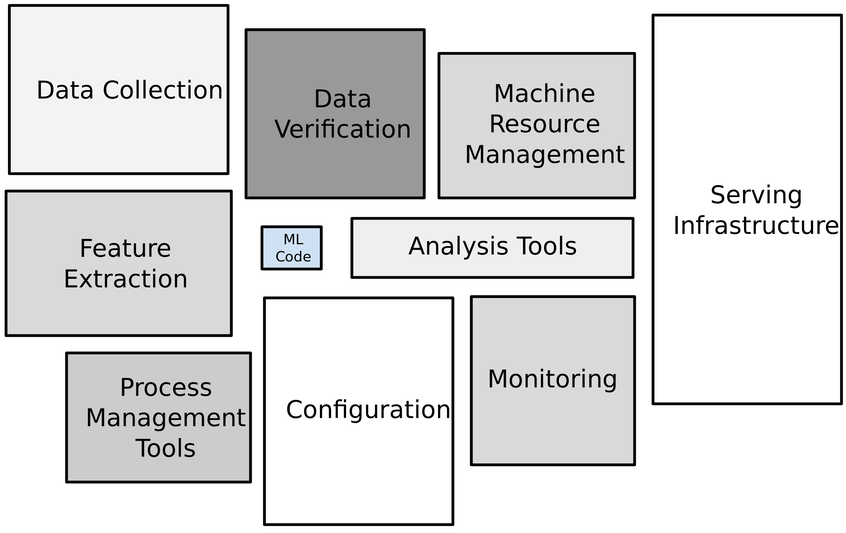
While we want enough dimensions to encode rich semantic relations, we also want an embedding space that is small enough to allow us to train our system more quickly. A useful embedding may be on the order of hundreds of dimensions. This is likely several orders of magnitude smaller than the size of your vocabulary for a natural language task.

Obtaining Embeddings

There are a number of ways to get an embedding, including a state-of-the-art algorithm created at Google: Word2vec

Production ML Systems

So far, Machine Learning Crash Course has focused on building ML models. However, as the following figure suggests, real-world production ML systems are large ecosystems of which the model is just a single part.



Real-world production ML system.

Fortunately, many of the components in the preceding figure are reusable. Furthermore, you don't have to build all the components in Figure 1 yourself. TensorFlow Extended (TFX) is an end-to-end platform for deploying production ML pipelines.

Static vs. Dynamic Training

Broadly speaking, there are two ways to train a model:

- A **static model** is trained offline. That is, we train the model exactly once and then use that trained model for a while.
- A **dynamic model** is trained online. That is, data is continually entering the system and we're incorporating that data into the model through continuous updates.

Broadly speaking, the following points dominate the static vs. dynamic training decision:

- Static models are easier to build and test.
- Dynamic models adapt to changing data. The world is a highly changeable place. Sales predictions built from last year's data are unlikely to successfully predict next year's results.

If your data set truly isn't changing over time, choose static training because it is cheaper to create and maintain than dynamic training. However, many information sources really do change over time, even those with features that you think are as constant as, say, sea level. The moral: even with static training, you must still monitor your input data for change.

For example, consider a model trained to predict the probability that users will buy flowers. Because of time pressure, the model is trained only once using a dataset of flower buying behavior during July and August. The model is then shipped off to serve predictions in production, *but is never updated.* The model works fine for several months, but then makes terrible predictions around Valentine's Day because user behavior during that holiday period changes dramatically.

Static vs. Dynamic Inference

You can choose either of the following inference strategies:

- **offline inference**, meaning that you make all possible predictions in a batch, using a MapReduce or something similar. You then write the predictions to an SSTable or Bigtable, and then feed these to a cache/lookup table.
- **online inference**, meaning that you predict on demand, using a server.

Data Dependencies

- The behavior of an ML system is dependent on the behavior and qualities of its input features. As the input data for those features changes, so too will your model. Sometimes that change is desirable, but sometimes it is not.
- In traditional software development, you focus more on code than on data. In machine learning development, although coding is still part of the job, your focus must widen to include data. For example, on traditional software development projects, it is a best practice to write unit tests to validate your code. On ML projects, you must also continuously test, verify, and monitor your input data.
- For example, you should continuously monitor your model to remove unused (or little used) features. Imagine a certain feature that has been contributing little or nothing to the model. If the input data for that feature abruptly changes, your model's behavior might also abruptly change in undesirable ways.

Reliability

- Some questions to ask about the reliability of your input data:
  - Is the signal always going to be available or is it coming from an unreliable source? For example:
  - Is the signal coming from a server that crashes under heavy load?
  - Is the signal coming from humans that go on vacation every August?

Versioning

- Some questions to ask about versioning:
  - Does the system that computes this data ever change? If so:
  - How often?
  - How will you know when that system changes?
- Sometimes, data comes from an upstream process. If that process changes abruptly, your model can suffer.
- Consider creating your own copy of the data you receive from the upstream process. Then, only advance to the next version of the upstream data when you are certain that it is safe to do so.

Necessity

- The following question might remind you of regularization:
  - Does the usefulness of the feature justify the cost of including it?
- It is always tempting to add more features to the model. For example, suppose you find a new feature whose addition makes your model slightly more accurate.

More accuracy certainly *sounds* better than less accuracy. However, now you've just added to your maintenance burden. That additional feature could degrade unexpectedly, so you've got to monitor it. Think carefully before adding features that lead to minor short-term wins.

Correlations

- Some features correlate (positively or negatively) with other features. Ask yourself the following question:
  - Are any features so tied together that you need additional strategies to tease them apart?

Feedback Loops

- Sometimes a model can affect its own training data. For example, the results from some models, in turn, are directly or indirectly input features to that same model.
- Sometimes a model can affect another model. For example, consider two models for predicting stock prices:
  - Model A, which is a bad predictive model.
  - Model B.
- Since Model A is buggy, it mistakenly decides to buy stock in Stock X. Those purchases drive up the price of Stock X. Model B uses the price of Stock X as an input feature, so Model B can easily come to some false conclusions about the value of Stock X stock. Model B could, therefore, buy or sell shares of Stock X based on the buggy behavior of Model A. Model B's behavior, in turn, can affect Model A, possibly triggering a tulip mania or a slide in Company X's stock

Fairness

Machine learning models are not inherently objective. Engineers train models by feeding them a data set of training examples, and human involvement in the provision and curation of this data can make a model's predictions susceptible to bias.

When building models, it's important to be aware of common human biases that can manifest in your data, so you can take proactive steps to mitigate their effects.

Reporting Bias

- Reporting bias occurs when the frequency of events, properties, and/or outcomes captured in a data set does not accurately reflect their real-world frequency.
- This bias can arise because people tend to focus on documenting circumstances that are unusual or especially memorable, assuming that the ordinary can "go without saying."
- *Eg: A sentiment-analysis model is trained to predict whether book reviews are positive or negative based on a corpus of user submissions to a popular website. The majority of reviews in the training data set reflect extreme opinions (reviewers who either loved or hated a book), because people were less likely to submit a review of a book if they did not respond to it strongly. As a result, the model is less able to correctly predict sentiment of reviews that use more subtle language to describe a book.*

Automation Bias

- Automation bias is a tendency to favor results generated by automated systems over those generated by non-automated systems, irrespective of the error rates of each.
- *Eg: Software engineers working for a sprocket manufacturer were eager to deploy the new "groundbreaking" model they trained to identify tooth defects, until the factory supervisor pointed out that the model's precision and recall rates were both 15% lower than those of human inspectors.*

Selection Bias

- Selection bias occurs if a data set's examples are chosen in a way that is not reflective of their real-world distribution. Selection bias can take many different forms:
- **Coverage bias:** Data is not selected in a representative fashion.
  Eg: A model is trained to predict future sales of a new product based on phone surveys conducted with a sample of consumers who bought the product. Consumers who instead opted to buy a competing product were not surveyed, and as a result, this group of people was not represented in the training data.
- **Non-response bias** (or **participation bias**): Data ends up being unrepresentative due to participation gaps in the data-collection process.

Eg: A model is trained to predict future sales of a new product based on phone surveys conducted with a sample of consumers who bought the product and with a sample of consumers who bought a competing product. Consumers who bought the competing product were 80% more likely to refuse to complete the survey, and their data was underrepresented in the sample.

- **Sampling bias**: Proper randomization is not used during data collection.
  Eg: A model is trained to predict future sales of a new product based on phone surveys conducted with a sample of consumers who bought the product and with a sample of consumers who bought a competing product. Instead of randomly targeting consumers, the surveyer chose the first 200 consumers that responded to an email, who might have been more enthusiastic about the product than average purchasers.

Group Attribution Bias

- Group attribution bias is a tendency to generalize what is true of individuals to an entire group to which they belong. Two key manifestations of this bias are:
- **In-group bias**: A preference for members of a group to which *you also belong*, or for characteristics that you also share.
  Eg: Two engineers training a résumé-screening model for software developers are predisposed to believe that applicants who attended the same computer-science academy as they both did are more qualified for the role.
- **Out-group homogeneity bias**: A tendency to stereotype individual members of a group to which *you do not belong*, or to see their characteristics as more uniform.
  Eg: Two engineers training a résumé-screening model for software developers are predisposed to believe that all applicants who did not attend a computer-science academy do not have sufficient expertise for the role.

Implicit Bias

- **Implicit bias** occurs when assumptions are made based on one's own mental models and personal experiences that do not necessarily apply more generally.
  Eg: An engineer training a gesture-recognition model uses a head shake as a feature to indicate a person is communicating the word "no." However, in some regions of the world, a head shake actually signifies "yes."

- A common form of implicit bias is **confirmation bias**, where model builders unconsciously process data in ways that affirm preexisting beliefs and hypotheses. In some cases, a model builder may actually keep training a model until it produces a result that aligns with their original hypothesis; this is called **experimenter's bias**.
  Eg: An engineer is building a model that predicts aggressiveness in dogs based on a variety of features (height, weight, breed, environment). The engineer had an unpleasant encounter with a hyperactive toy poodle as a child, and ever since has associated the breed with aggression. When the trained model predicted most toy poodles to be relatively docile, the engineer retrained the model several more times until it produced a result showing smaller poodles to be more violent.

Fairness: Identifying Bias

- Missing Feature Values
- Unexpected Feature Values
- Data Skew

Fairness: Evaluating for Bias

When evaluating a model, metrics calculated against an entire test or validation set don't always give an accurate picture of how fair the model is.

Consider a new model developed to predict the presence of tumors that is evaluated against a validation set of 1,000 patients' medical records. 500 records are from female patients, and 500 records are from male patients. The following confusion matrix summarizes the results for all 1,000 examples:

True Positives (TPs): 16	**False Positives (FPs): 4**
False Negatives (FNs): 6	**True Negatives (TNs): 974**

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{16}{16 + 4} = 0.800$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{16}{16 + 6} = 0.727$$

These results look promising: precision of 80% and recall of 72.7%. But what happens if we calculate the result separately for each set of patients? Let's break out the results into two separate confusion matrices: one for female patients and one for male patients.

Female Patient Results

True Positives (TPs): 10	False Positives (FPs): 1
False Negatives (FNs): 1	True Negatives (TNs): 488

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{10}{10 + 1} = 0.909$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{10}{10 + 1} = 0.909$$

Male Patient Results

True Positives (TPs): 6	False Positives (FPs): 3
False Negatives (FNs): 5	True Negatives (TNs): 486

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{6}{6 + 3} = 0.667$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{6}{6 + 5} = 0.545$$

When we calculate metrics separately for female and male patients, we see stark differences in model performance for each group.

Female patients:

- Of the 11 female patients who actually have tumors, the model correctly predicts positive for 10 patients (recall rate: 90.9%). In other words, **the model misses a tumor diagnosis in 9.1% of female cases**.
- Similarly, when the model returns positive for tumor in female patients, it is correct in 10 out of 11 cases (precision rate: 90.9%); in other words, **the model incorrectly predicts tumor in 9.1% of female cases**.

Male patients:

- However, of the 11 male patients who actually have tumors, the model correctly predicts positive for only 6 patients (recall rate: 54.5%). That means **the model misses a tumor diagnosis in 45.5% of male cases**.
- And when the model returns positive for tumor in male patients, it is correct in only 6 out of 9 cases (precision rate: 66.7%); in other words, **the model incorrectly predicts tumor in 33.3% of male cases**.

We now have a much better understanding of the biases inherent in the model's predictions, as well as the risks to each subgroup if the model were to be released for medical use in the general population.

Additional Fairness Resources

Fairness is a relatively new subfield within the discipline of machine learning. To learn more about research and initiatives devoted to developing new tools and techniques for identifying and mitigating bias in machine learning models, check out Google's Machine Learning Fairness resources page.

Effective ML Guidelines

- Keep your first model simple.
- Focus on ensuring data pipeline correctness.
- Use a simple, observable metric for training & evaluation.
- Own and monitor your input features.

- Treat your model configuration as code: review it, check it in.
- Write down the results of all experiments, especially "failures."