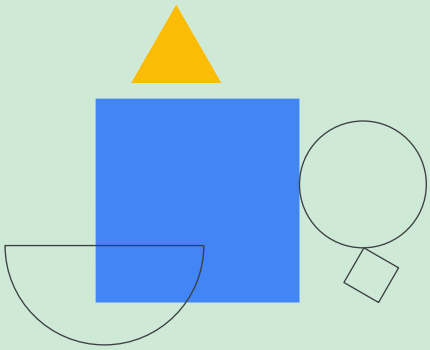
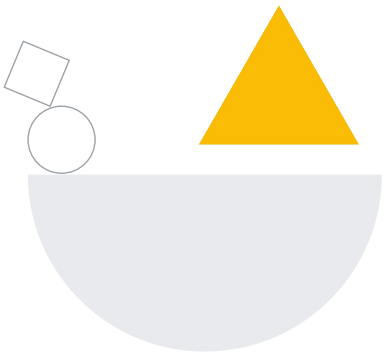
The Art and Science of Machine Learning (Additional lecture notes)

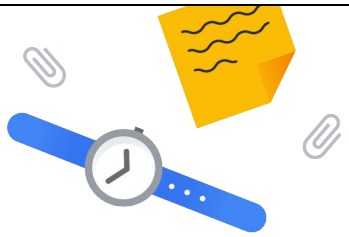


A Pinch of Science



Welcome to A Pinch of Science.

In this module, I will start to introduce the science along with the art of machine learning. We're first going to talk about how to perform regularization for sparsity so that we can have simpler, more concise models. Then we're going to talk about logistic regression and learning how to determine performance.



Module agenda

01 **Regularization for sparsity**

02 Logistic regression

Zeroing out coefficients can help with performance, especially with large models and sparse inputs

$$L(w,D) + \lambda \sum^{n} |w|$$

L2 regularization only makes weights small, not zero.

Action	Impact
• Fewer coefficients to store/load • Fewer multiplications needed	• Reduce memory, model size • Increase prediction speed

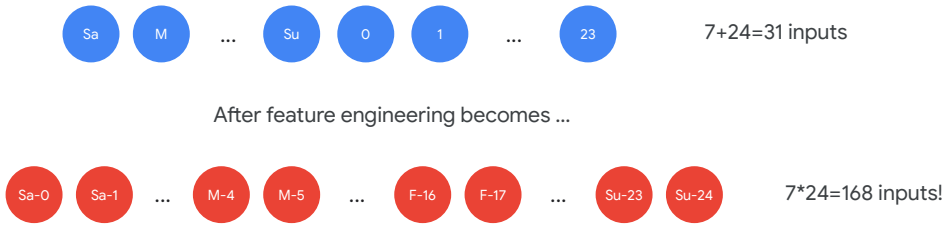
Earlier in the course we learned about L2 regularization which added a sum of the squared parameter weights term to the loss function. This was great at keeping weights small, having stability, and a unique solution, but it can leave the model unnecessarily large and complex since all of the features may still remain albeit with small weight.

Using something instead called L1 regularization adds the sum of the absolute value of the parameters weights to the loss function which tends to force the weights of not very predictive features to zero. This acts as a built-in feature selector by killing off bad features and leaving only the strongest in the model.

This sparse model has many benefits. First with fewer coefficients to store and load there is a reduction in storage and memory needed with a much smaller model size, which is especially important for embedded models. Also with fewer features there are a lot fewer mult-adds which not only leads to increased training speed but more importantly increased prediction speed.

L(w,D) + \lambda\sum_{ }^{n} |w|

Feature crosses lead to lots of input nodes, so having zero weights is especially important



Many machine learning models already have enough features as it is. For instance, let's say that I have data that contains the datetime of orders being placed. A first order model would probably include 7 features for the days of the week and 24 features for hours of the day plus possibly many other features. Therefore the day of the week plus hour of the day is already 31 inputs with just that.

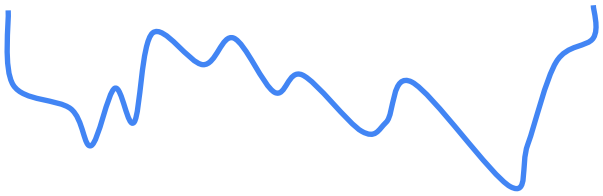
Now what if we want to look at the second order effects of day of the week crossed with hour of the day. That is another 168 inputs in addition to our 31 plus others for a grand total now of almost 200 features just for that one datetime field plus whatever other features we are using.

If we cross this with the one hot encoding for U.S. state for example, the triple cartesian product is already at 8,400 features with many of them probably being very sparse of mostly zeroes. Hopefully this makes clear why built-in feature selection through L1 regularization can be a very good thing.

What strategies can we use to remove feature coefficients that aren't useful besides L1 regularization perhaps?

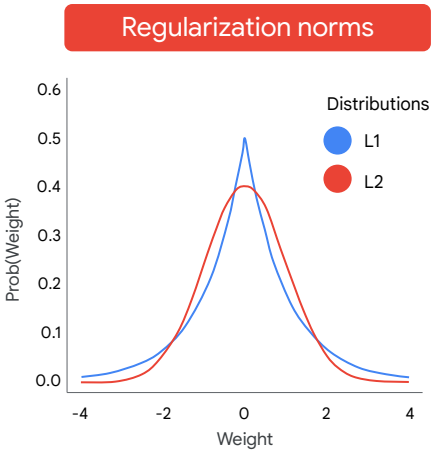
We could include using simple counts of which features occur with non-zero values.

L0-norm (the count of non-zero weights) is an NP-hard, non-convex optimization problem

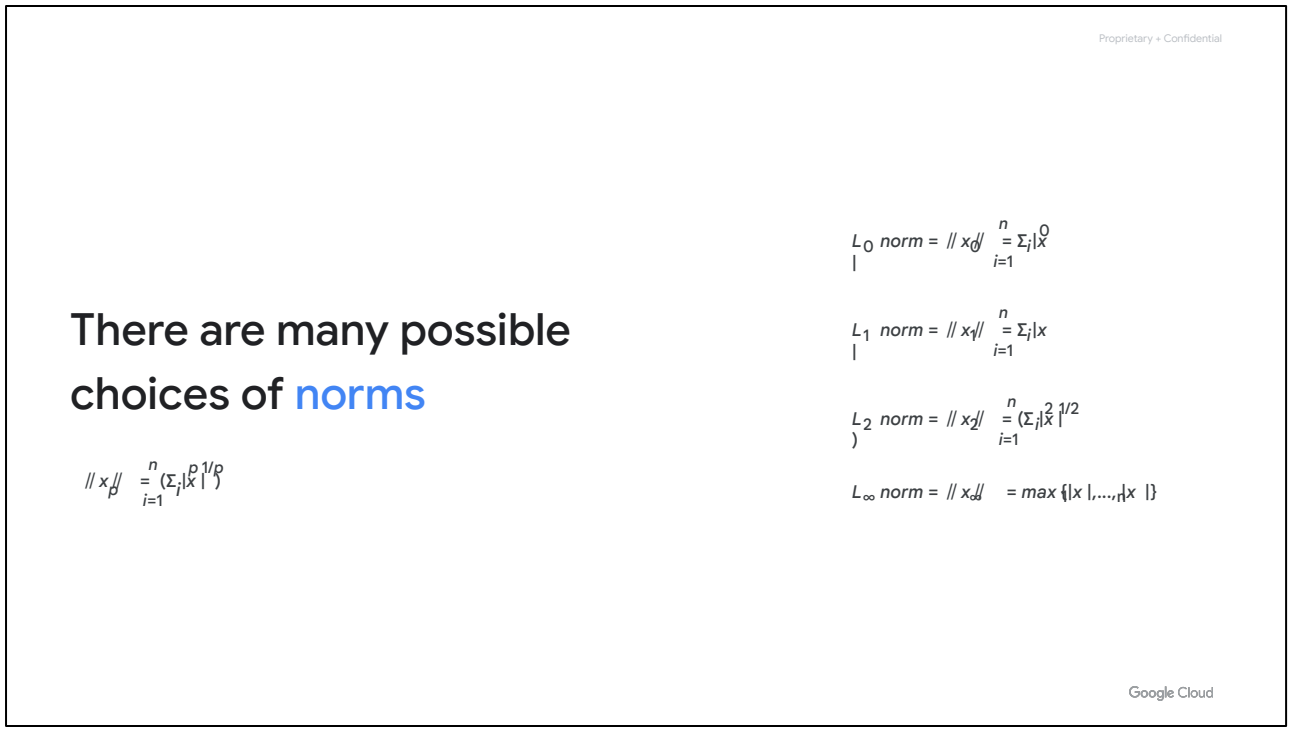


The L0 norm is simply the count of the non-zero weights and optimizing for this norm is an NP-hard non-convex optimization problem. This diagram illustrates what a non-convex optimization error surface might look like. As you can see there are many local peaks and valleys and this is just a simple 1 dimensional example. You pretty much have to explore lots and lots of starting points with gradient descent, making this an NP-hard problem to solve completely.

L_1 norm (sum of absolute values of the weights) is convex and efficient; it tends to encourage sparsity in the model



Thankfully the L1 norm, just like the L2 norm, is convex, but it also encourages sparsity in the model. In this figure, the probability distributions of the L1 and L2 norms are plotted. Notice how the L2 norm has a much smoother peak at zero which results in the magnitudes of weights being closer to zero. However, you'll notice the L1 norm is more of a cusp centered at zero, therefore much more of the probability is exactly at zero than the L2 norm.



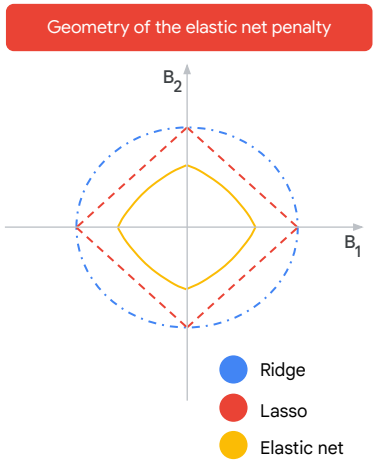
There are an infinite number of norms which are generalized by the p-norm. Some other norms are the L0 norm that we already covered which is the count of non-zero values in a vector and the L-infinity norm which is the maximum absolute value of any value in a vector.

In practice though, usually the L2 norm provides more generalizable models than the L1 norm however we will end up with much more complex, heavy models if we use L2 instead of L1. This happens because often features have high correlation with each other and L1 regularization will just choose one of them and throw away the other. Whereas L2 regularization will keep both features and keep their weight magnitudes small. So with L1, you can end up with a smaller model, but it may be less predictive. Is there any way to get the best of both worlds?

||x||_p=\left (\sum_{i=1}^{n} |x_i|^p \right)^{1/p}
L_0\ norm=||x||_0=\sum_{i=1}^{n} |x_i|^0
L_1\ norm=||x||_1=\sum_{i=1}^{n} |x_i|
L_2\ norm=||x||_2=\left (\sum_{i=1}^{n} |x_i|^2 \right)^{1/2}
L_\infty\ norm=||x||_\infty=max\left \{ |x_1|,...,|x_n| \right \}

Elastic nets combine the feature selection of L1 regularization with the generalizability of L2 regularization

$$L(w,D) + \lambda_1 \sum^{n} |w| + \lambda_2 \sum^{n} w^2$$



The elastic net is just the linear combination of the L1 and L2 regularization penalties. This way you get the benefits of sparsity for really poor predictive features while also keeping decent and great features with smaller weights to provide good generalization. The only tradeoff is now there are two instead of one hyperparameters to tune with the two different lambda regularization parameters.

L(w,D) + \lambda _1\sum_{ }^{n} |w|+\lambda _2 \sum_{ }^{n}w^2

Quiz: L1 Regularization

What does L1 regularization tend to do to a model's low predictive features' parameter weights?

A. Have small magnitudes

B. Have all positive values

C. Have zero values

D. Have large magnitudes

Question: What does L1 regularization tend to do to a model's low predictive features' parameter weights?

Quiz: L1 Regularization

What does L1 regularization tend to do to a model's low predictive features' parameter weights?

A. Have small magnitudes

B. Have all positive values

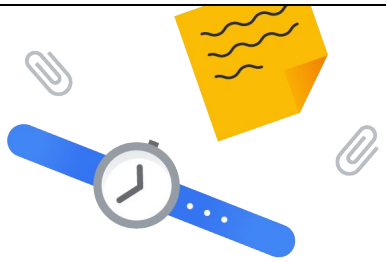
C. Have zero values

D. Have large magnitudes

Answer: The correct answer is have zero values. Whenever we are doing regularization techniques, we are adding a penalty term to the loss function, or in general the objective function, so that it doesn't over-optimize our decision variables or parameter weights. We choose the penalty terms based on prior knowledge, function shape, etc. L1 Regularization has been shown to induce sparsity to the model and due to its probability distribution having a high peak at 0, most weights except for the highly predictive ones will be shifted from their non-regularized values to zero.

L2 regularization would be used for having small magnitudes and its negative would be used for having large magnitudes which are both incorrect.

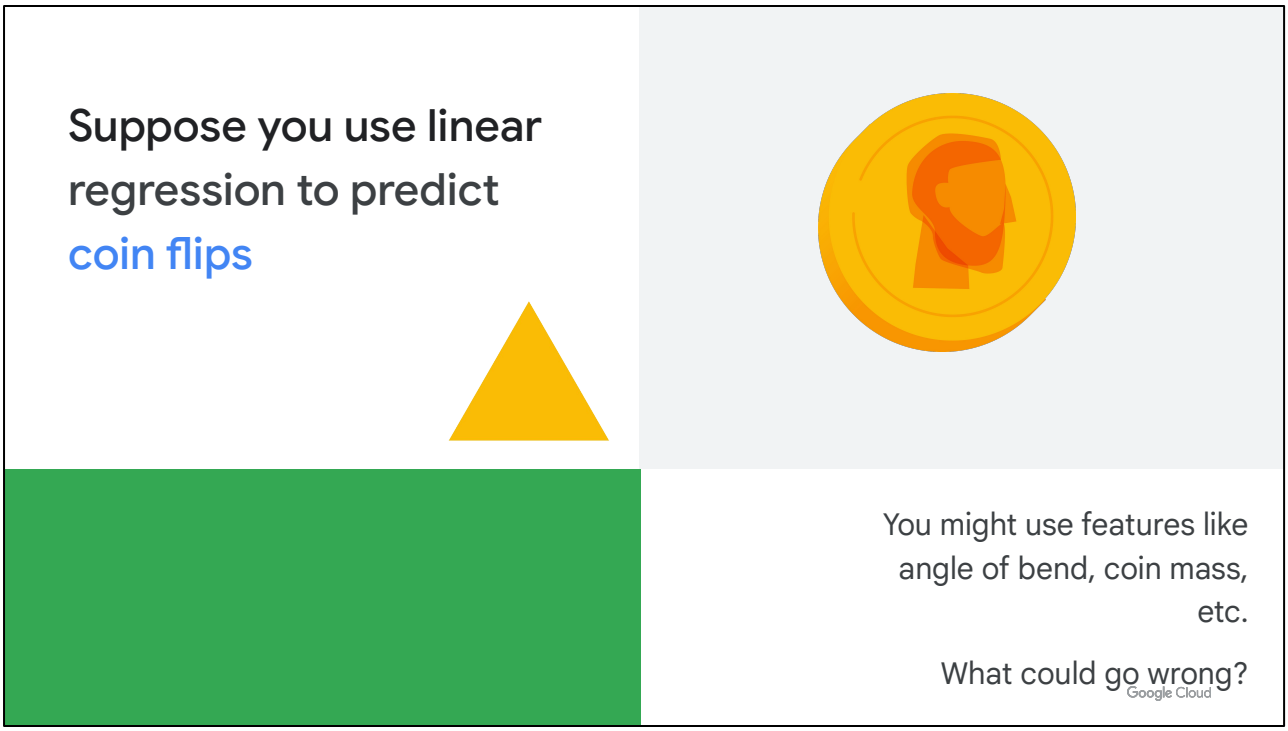
Having all positive values would be like adding many additional constraints to the optimization problem bounding all decision variables to be greater than 0 which is also not L1 regularization.



Module agenda

01 Regularization for sparsity

02 **Logistic regression**



Suppose we want to predict the outcomes of coin flips. We all know that for a fair coin, the expected value is 50% heads and 50% tails. What if we had instead an unfair coin like with a bend in it. Let's say we want to generalize coin flip prediction to all coins: fair and unfair, big and small, heavy and light, etc. What features could we use to predict whether a flip would be heads or tails?

Perhaps we could use the angle of the bend because it distributes X% of mass in the other dimension and/or creates a difference in rotation due to air resistance or center of mass. The mass of the coin might also be a good feature to know, as well as size properties such as diameter, thickness, etc. We could do some feature engineering on this to get the volume of the coin and furthermore the density. Maybe the type of material or materials the coin is composed of would be useful information.

These features would be pretty easy to measure however they are only one side of the coin, pun intended, of the coin flip, the rest comes down to the action of the flip itself such as how much linear and angular velocity the coin was given, the angle of launch, the angle of what it lands on, wind speed, etc. These might be a bit harder to measure.

Now that we have all of these features, what's the simplest model we could use to predict heads or tails? Linear regression of course. What could go wrong with this choice though?

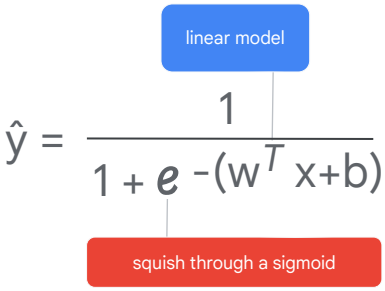
Our labels are heads or tails, or thought of in another way heads or not heads which

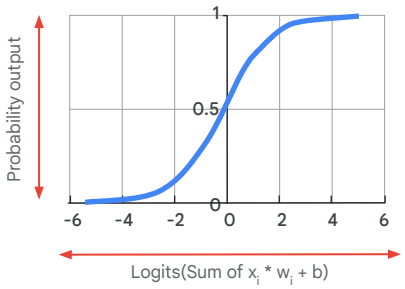
we can represent with a one-hot encoding of 1 for heads and 0 for not-heads. But if we use linear regression with the standard mean squared error loss function, our predictions could end up being outside the range of (0, 1). What does it mean if we predict 2.75 for the coin flip state? That makes no sense.

A model that minimizes squared error is under no constraint to treat this as a probability in (0, 1), but this is what we need here. In particular, you could imagine a model that predicts values less than 0 or greater than 1 for some new examples. This would mean we can't use this model as a probability.

Simple tricks like capping the predictions at 0 or 1 would introduce bias. So, we need something else -- in particular, a new loss function.

Logistic Regression: transform linear regression by a sigmoid activation function





Converting this from linear regression to logistic regression can solve this dilemma. From an earlier course of ours, we went through the history of ML and introduced the Sigmoid activation function. Let's take a deeper look into that now.

The sigmoid activation function essentially takes the weighted sum, w transpose x plus b, from a linear regression and instead of just outputting that and then calculating the mean squared error loss, we change the activation function from linear to sigmoid which takes that as an argument and squashes it smoothly between 0 and 1. The input into the sigmoid, normally the output of a linear regression, is called the logit. So we are performing a non-linear transformation of our linear model.

Notice how the probability asymptotes to zero when the logits go to negative infinity and to one when the logits go to positive infinity. What does this imply for training? Unlike mean squared error, the sigmoid never guesses 1.0 or 0.0 probability. This means, that in gradient descent's constant drive to get the loss closer and closer to zero, it will drive the weights closer and closer to plus or minus infinity, in the absence of regularization, which can lead to problems.

\hat{y}=\frac{1}{1+e^{-(\mathbf{w}^{T}\mathbf{x}+b)}}

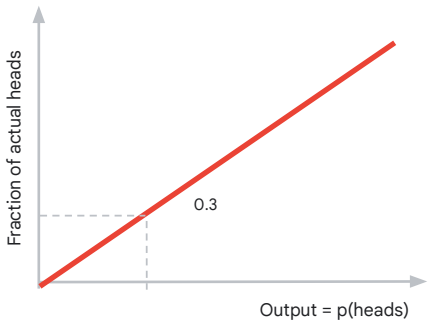
The output of Logistic Regression is a calibrated probability estimate

Useful because we can cast binary classification problems into probabilistic problems:

Will customer buy item?

becomes

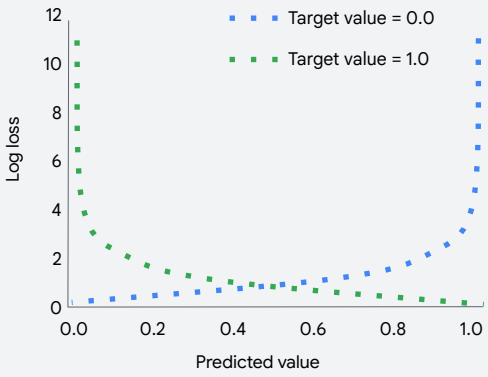
Predict the probability that customer buys item



First though, how can we interpret the output of a sigmoid? Is it just some function that's range is 0 to 1, of which there are many, or is it something more? The good news is it is something more, it is a calibrated probability estimate. Beyond just the range, the sigmoid function is the cumulative distribution function of the logistic probability distribution whose quantile function is the inverse of the logit which models the log odds. Therefore, mathematically the outputs of a sigmoid can be considered probabilities.

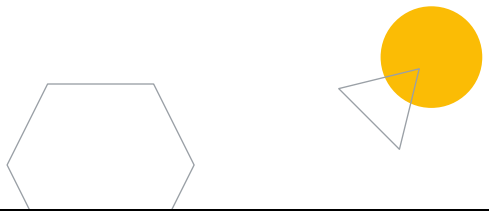
In this way, we can think of calibration as the fact that the outputs are real world values like probabilities. This is in contrast to uncalibrated outputs like an embedding vector, which is internally informative, but the values have no real world correlation. Lots of output activation functions, in fact an infinite number, could give you a number between 0 and 1 but only the sigmoid is proven to be a calibrated estimate of the training dataset probability of occurrence.

Using this fact about the sigmoid activation function, we can cast binary classification problems into probabilistic problems. For instance, instead of a model just predicting a yes or no, such as will a customer buy an item, it can now predict the probability that a customer buys an item. This paired with a threshold can provide a lot more predictive power than just a simple binary answer.



Typically, use cross-entropy (related to Shannon's information theory) as the error metric

Less emphasis on errors where the output is relatively close to the label.

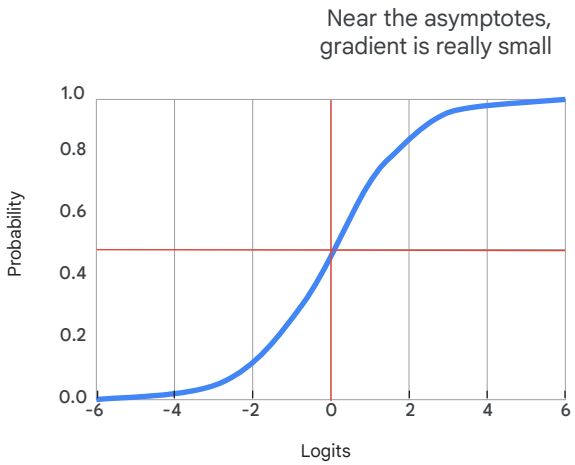


$$LogLoss = \sum_{(x,y) \in D} -y\log(\hat{y}) - (1 - y)\log(1 - \hat{y})$$

However, regularization is important in logistic regression because driving the loss to zero is difficult and dangerous.

First, as gradient descent seeks to minimize cross entropy it pushes output values closer to 1 for positive labels and closer to 0 for negative labels. Due to the equation of the sigmoid, the function asymptotes to 0 when the logit is negative infinity and to 1 when the logit is positive infinity. To get the logits to negative or positive infinity, the magnitude of the weights is increased and increased leading to numerical stability problems: overflows and underflows. This is dangerous and can ruin our training.

Regularization is important in logistic regression because driving the loss to zero is difficult and dangerous



Also, near the asymptotes, as you can see from the graph, the sigmoid function becomes flatter and flatter. This means that the derivative is getting closer and closer to zero. Since we use the derivative in backpropagation to update the weights, it is important for the gradient not to become zero, or else training will stop. This is called saturation when all activations end up in these plateaus which leads to a vanishing gradient problem and makes training difficult.

There is also a potentially useful insight here. Imagine you assign a unique id for each example, and map each id to its own feature. If you use un-regularized logistic regression, this will lead to absolute overfitting, as the model tries to drive loss to zero on all examples and never gets there, the weights for each indicator feature will be driven to +inf or -inf. This can happen in practice in high dimensional data with feature crosses. Often there's a huge mass of rare crosses that happens only on one example each. So how can we protect ourselves from overfitting?

Also, near the asymptotes, as you can see from the graph, the sigmoid function becomes flatter and flatter. This means that the derivative is getting closer and closer to zero. Since we use the derivative in backpropagation to update the weights, it is important for the gradient not to become zero, or else training will stop. This is called saturation when all activations end up in these plateaus which leads to a vanishing gradient problem and makes training difficult.

There is also a potentially useful insight here. Imagine you assign a unique id for each example, and map each id to its own feature. If you use un-regularized logistic

regression, this will lead to absolute overfitting, as the model tries to drive loss to zero on all examples and never gets there, the weights for each indicator feature will be driven to +inf or -inf. This can happen in practice in high dimensional data with feature crosses. Often there's a huge mass of rare crosses that happens only on one example each. So how can we protect ourselves from overfitting?

Quiz: Logistic Regression Regularization

Why is it important to add regularization to logistic regression?

A. Helps stops weights being driven to +/- infinity

B. Helps logits stay away from asymptotes which can halt training

C. Transforms outputs into a calibrated probability estimate

D. Both A & B

E. Both A & C

Question: Which of these is important when performing logistic regression?

Quiz: Logistic Regression Regularization

Why is it important to add regularization to logistic regression?

A. Helps stops weights being driven to +/- infinity

B. Helps logits stay away from asymptotes which can halt training

C. Transforms outputs into a calibrated probability estimate

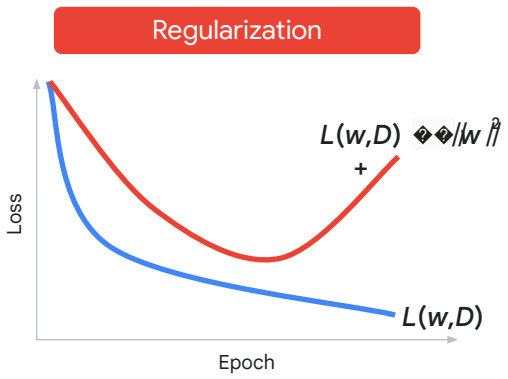
D. Both A & B

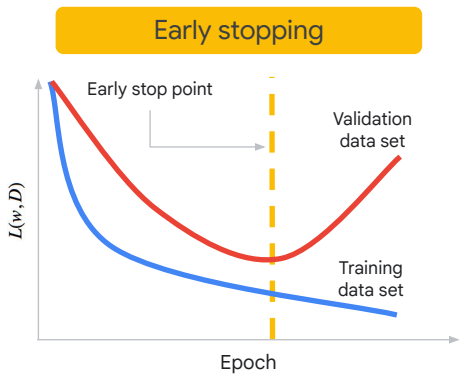
E. Both A & C

Answer: The correct answer is both A & B. Adding regularization to logistic regression helps keep the model simpler by having smaller parameter weights. This penalty term added to the loss function makes sure that cross entropy through gradient descent doesn't keep pushing the weights from going closer and closer to plus or minus infinity and casuing numerical issues. Also, with now smaller logits, we can now stay in the less flat portions of the sigmoid function making our gradients less close to zero and thus allowing weight updates and training to continue.

C is incorrect and therefore so is E because regularization does not transform the outputs into a calibrated probability estimate. The great thing about logistic regression is that it already outputs a calibrated probability estimate since the sigmoid function is the cumulative distribution function of the logistic probability distribution. This allows us to actually predict probabilities instead of just binary answers like yes or no, true or false, buy or sell, etc.

Often we do both regularization and early stopping to counteract overfitting





To counteract overfitting we often do both regularization and early stopping.

For regularization, model complexity increases with large weights, and so as we tune and start to get larger and larger weights for rarer and rarer scenarios, we end up increasing the loss, so we stop. L2 regularization will keep the weight values smaller and L1 regularization will make the model sparser by dropping poor features. To find the optimal L1 and L2 hyperparameter choices during hyperparameter tuning, you are searching for the point in the validation loss function where you obtain the lowest value. At that point any less regularization increases your variance, starts overfitting, and hurts your generalization and any more regularization increases your bias, starts underfitting, and hurts your generalization.

Early stopping stops training when overfitting begins. As you train your model, you should evaluate your model on your validation dataset every so many steps, epochs, minutes, etc. As training continues both the training error and the validation error should be decreasing but at some point the validation error might begin to actually increase! It is at this point that the model is beginning to memorize the training dataset and lose its ability to generalize to the validation dataset, and most importantly, to the new data that we eventually want to use this model for. Using early stopping would stop the model at this point and then backup and use the weights from the previous step before it hit the validation error inflection point. Here, the loss is just L(w,D) i.e. no regularization term. Interestingly, early stopping is an approximate equivalent of L2 regularization and is often used in its place because it is computationally cheaper.

Fortunately, in practice, we always use both explicit regularization (L_1 and L_2), and also some amount of early stopping regularization. Even though L2 regularization and early stopping seem a bit redundant, for real world systems, you may not quite choose the optimal hyperparameters and thus early stopping can help fix that choice for you.

In many real-world problems, the probability is not enough; we need to make a binary decision

Send the mail to spam folder or not?

Approve the loan or not?

Which road should we route the user through?

Choice of threshold is important and can be tuned.

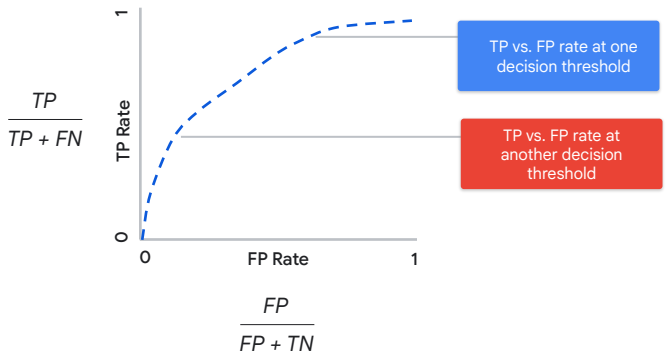
It's great that we can obtain a probability from our logistic regression model, however at the end of the day, sometimes users just want a simple decision to be made for them for their real world problems. Should the email be sent to the spam folder or not? Should the loan be approved or not? Which road should we route the user through?

How can we use our probability estimate to help the tool, using our model, to make a decision? We choose a threshold!

A simple threshold of a binary classification problem would be all probabilities less than or equal to 50% should be a no and all probabilities greater than 50% should be a yes. However, for certain real world problems we may want a different split like 60/40, 20/80, 99/1 etc. depending on how we want our balance of our type I and type II errors or in other words our balance of false positives and false negatives.

For binary classification, we will have four possible outcomes: true positives, true negatives, false positives, and false negatives. Combinations of these values can lead to evaluation metrics like precision which is the number of true positives divided by all positives and recall which is the number of true positives divided by the sum of true positives and false negatives which gives the sensitivity or true positive rate. You can tune your choice of threshold to optimize the metric of your choice. Is there an easy way to help us do this?

Use the ROC curve to choose the decision threshold based on decision criteria



A Receiver Operating Characteristic curve, or ROC curve for short, shows how a given model's predictions create different true positive vs. false positive rates when different decision thresholds are used.

As we lower the threshold, we are likely to have more false positives, but will also increase the number of true positives we find. Ideally, a perfect model would have zero false positives and zero false negatives which plugging that into the equations would give a true positive rate of 1 and a false positive rate of 0.

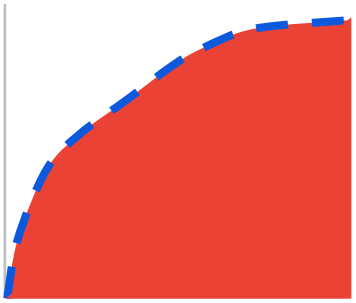
To create a curve, we would pick each possible decision threshold and re-evaluate. Each threshold value creates a single point, but by evaluating many thresholds eventually a curve is formed. Fortunately, there's an efficient sorting-based algorithm to do this.

Each model will create a different ROC curve, so how can we use these curves to compare the relative performance of our models when we don't know exactly what decision threshold we want to use?

The Area-Under-Curve (AUC) provides an aggregate measure of performance across all possible classification thresholds

AUC helps you choose between models when you don't know what decision threshold is going to be ultimately used.

"If we pick a random positive and a random negative, what's the probability my model scores them in the correct relative order?"



We can use the area under the curve as an aggregate measure of performance across all possible classification thresholds. AUC helps you choose between models when you don't know what decision threshold is going to be ultimately used. It is like asking "if we pick a random positive and a random negative, what's the probability my model scores them in the correct relative order?"

The nice thing about AUC is that it's scale invariant and classification threshold invariant. People like to use it for those reasons. People sometimes also use AUC for the precision-recall curve, or more recently precision-recall-gain curves which just use different combinations of the four prediction outcomes as metrics along the axes.

However, treating this only as an aggregate measure can mask some effects. For example, a small improvement in AUC might come by doing a better job of ranking "very unlikely negatives" as "even still yet more unlikely", which is fine but potentially not materially beneficial.

Logistic Regression predictions should be unbiased

average of predictions == average of observations

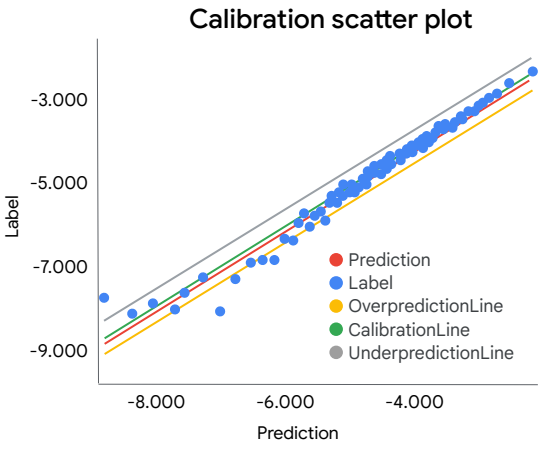
Look for bias in slices of data, this can guide improvements.

When evaluating our logistic regression models, we need to make sure predictions should be unbiased. When we talk about bias in this sense we are not talking about the bias term in the model's linear equation. Instead we mean, there should not be an overall shift in either the positive or negative direction. A simple way to check the prediction bias is to compare the average value of predictions made by the model over a dataset to the average value of the labels in that dataset. If they are not relatively close then you might have a problem.

Bias is like a canary in a mine where we can use it as an indicator of something being wrong. If you have bias, you definitely have a problem, but even zero bias alone does not mean everything in your system is perfect, but it is a great sanity check. If you have bias, you could have an incomplete feature set, a buggy pipeline, a biased training sample, etc. You can look for bias in slices of data which can help guide improvements of removing bias from your model. Let's look at an example of how you can do that.

Use calibration plots of bucketed bias to find slices where your model performs poorly

Each dot represents many examples in the same bucketed prediction range.



Here's a calibration plot from the sibyl experiment browser. You'll notice that this is on a log / log scale, as we're comparing the bucketized log-odds predicted to the bucketized log-odds observed.

You'll note that things are pretty well calibrated in the moderate range, but the extreme low end is pretty bad.

This can happen when parts of the data space is not well represented, or because of noise, or because of overly strong regularization.

The bucketing can be done in a couple of ways. You can bucket by linearly breaking up the target predictions, or you can bucket by quantiles.

Why do we need to bucket prediction to make calibration plots when predicting probabilities?

For any given event, the true label is either 0 or 1, for example, not clicked or clicked. But our prediction values will always be a probabilistic guess somewhere in the middle, like 0.1 or 0.33. For any individual example, we're always off. But if we group enough examples together, we'd like to see that on average the sum of the true 0's and 1's is about the same as the mean probability we're predicting.

Quiz: Logistic Regression

Which of these is important when performing logistic regression?

A. Adding regularization

B. Choosing a tuned threshold

C. Checking for bias

D. All of the above

Question: Which of these is important when performing logistic regression?

Quiz: Logistic Regression

Which of these is important when performing logistic regression?

A. Adding regularization

B. Choosing a tuned threshold

C. Checking for bias

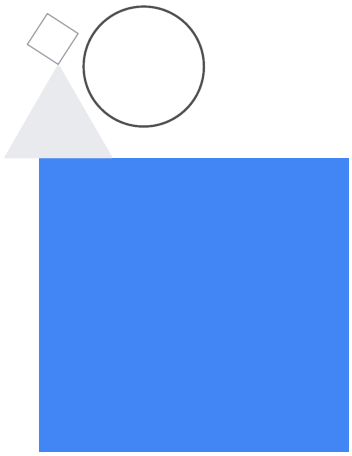
D. All of the above

Answer: The correct answer is all of the above. It is extremely important that our model generalizes so that we have the best predictions on new data which is the entire reason we created the model to begin with. To help do this, it is important that we do not overfit our data, therefore adding penalty terms to the objective function like with L1 regularization for sparsity and L2 regularization for keeping model weights small and adding early stopping can help in this regard.

It is also important to choose a tuned threshold for deciding what decisions to make with your probability estimate outputs to minimize or maximize the business metric that is important to you. If this isn't well defined, then we can use more statistical means such as calculating the number of true and false positives and negatives and combining them into different metrics such as the true and false positive rates. We can then repeat this process for many different thresholds and then plot the area under the curve, or AUC, to come up with a relative aggregate measure of model performance.

Lastly, it is important that our predictions are unbiased and even if there isn't bias, we should still be dilligent to make sure our model is performing well. We can begin looking for bias by making sure that the average of the predictions is very close to the average of observations. A helpful way to find where bias might be hiding is to look at slices of data and use something like a calibration plot to isolate the problem areas for further refinement.

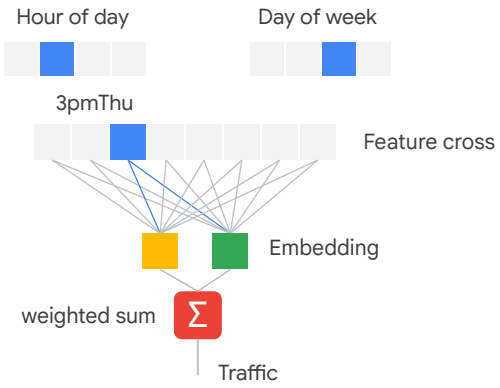
Embeddings



Welcome to Embeddings.

In this module, we will go back and revisit an important concept called embeddings.

Creating an embedding column from a feature cross



Embeddings are everywhere in modern machine learning, and they are not limited to feature crosses or even to structured data. In fact, you will use them in image models and quite a bit in text models.

Let's do a quick recap of embeddings the way we understand them.

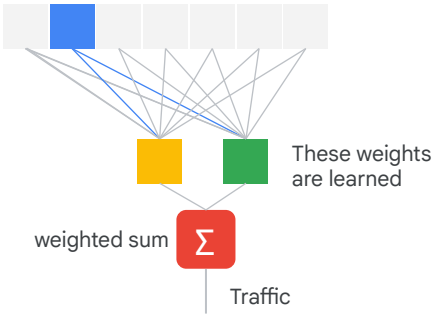
We said that we might be building a machine learning model to predict something about the traffic -- perhaps it's the time before the next vehicle arrives at an intersection -- and we have a number of inputs into our model. We looked specifically at categorical inputs, hour-of-day and day-of-week.
We said that the ML model would be greatly improved if, instead of treating the hour-of-day and day-of-week as independent inputs, we essentially concatenated them to create a feature cross. We said that if we used a large number of hashbuckets when doing this feature cross, we could be relatively confident that each of the buckets contained only one hour-day combination.

This was the point at which we introduced embeddings. We said that if, instead of one-hot encoding the feature cross and using as is-is, we could pass it through a dense layer and then train the model to predict traffic as before.

This dense layer, shown by the yellow and green nodes, here creates an embedding.

The embeddings are real-valued numbers because they are a weighted sum of the feature crossed values.

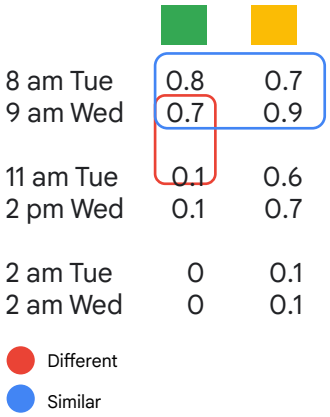
The weights in the embedding column are learned from data



The thing to realize is that the weights that go into the embedding layer (the yellow and green nodes) are learned from the data.

The point is that by training these weights on a dataset, to solve a useful problem, something neat happens …
The feature cross of day-hour has 168 unique values, but we are forcing it to be represented with just two real-valued numbers. So ...

The model learns how to embed the feature cross in lower-dimensional space



the model learns how to embed the feature cross in lower-dimensional space.
We suggested that perhaps the green box tends to capture pedestrian traffic while the yellow tends to capture automobile traffic.
But it doesn't matter what exactly those two dimensions are capturing. The important thing is that all the information in the hour-of-day and day-of-week as it pertains to traffic at city intersections is shoehorned into just two numbers.

If you do this on a large enough and good enough dataset, these numbers have one very useful property:
Times that are similar in terms of traffic get real-valued numbers that are close together.
And times are different in terms of traffic get real-valued numbers that are different.

Embedding a feature cross in TensorFlow

```
import tf.feature_column as fc

day_hr = fc.crossed_column(
   [dayofweek, hourofday],
   24x7 )

day_hr_em = fc.embedding_column(
   day_hr,
   2 )
```


We then looked at how to create an embedding in TensorFlow.

To create an embedding, use the embedding_column method in tf.feature_column
And pass in the categorical column you want to embed.
This works with any categorical column, not just a feature cross.

You do an embedding of any categorical column.

Transfer Learning of embeddings from similar ML models

```
import tf.feature_column as fc

day_hr = fc.crossed_column(
   [dayofweek, hourofday],
   7 * 24)

day_hr_em = fc.embedding_column(
   day_hr,
   2,
   ckpt_to_load_from='london/*ckpt-1000*',
   tensor_name_in_ckpt='dayhr_embed',
   trainable=False
 )
```


Finally, we glanced quickly at how you could take the embeddings that you learned on problem and apply to another similar ML problem.

Perhaps you learned how to represent hour-of-day and day-of-week with two real-valued numbers by training on traffic data in London.
As a quick start, you can use the same weights to jump start your Frankfurt model.

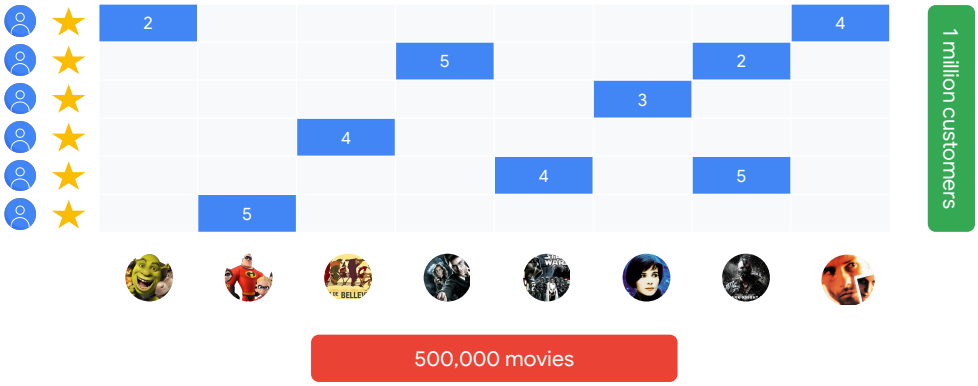
Transfer Learning of embeddings from similar ML models

First Layer	the feature cross
Second Layer	a mystery box labeled latent factor
Third Layer	the embedding
Fourth Layer	one side: image of traffic
Second Side	image of people watching TV

You might even be able to use the embedding that you learned on the traffic problem to predict the viewership of a TV show!
The idea being that both street traffic and TV viewership depend on the same *latent* factor, namely, are the people in the city on the move or are they at home or at work?

Transfer Learning might work on seemingly very different problems as long as they share the same latent factors.

How do you recommend movies to customers?

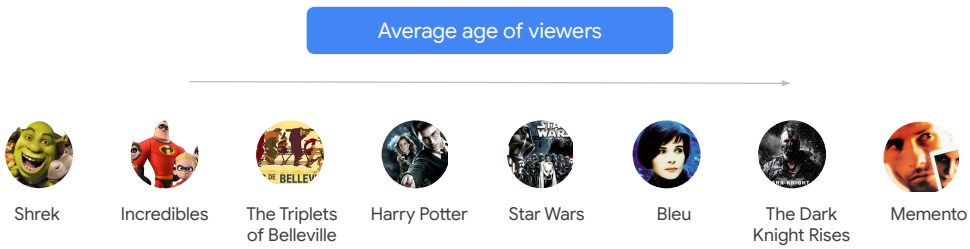


So, what's our input dataset? Our input dataset, if we represent it as a matrix, is 1 million rows and 500,000 columns.
The numbers in the diagram denote movies that customers have watched and rated.

What we need to do is to figure out the rest of the matrix.

To solve this problem some method is needed to determine which movies are similar to each other.

One approach is to organize movies by similarity (1D)



One approach is to organize movies by similarity using some attribute of the movies.

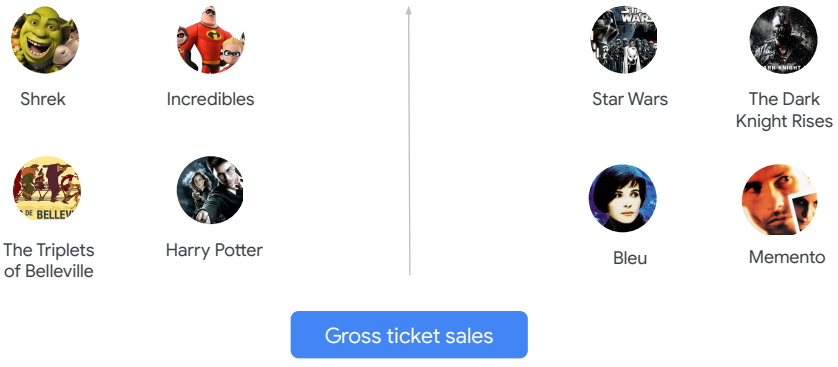
For example, we might look at the average age of the audience and put the movies on a line.

So, the cartoons and animated movies show up on the left-hand side and the darker, adult-oriented movies show up to the right.
Then, we can say that if you liked the Incredibles, perhaps you are a child or you have a young child, and so we can recommend Shrek to you.

Blue and Memento are arthouse movies whereas Star Wars and the Dark Knight Rises are both blockbusters.
If someone watched and like Bleu, they are more likely to like Memento than a movie about Batman.
Similarly, someone who watched and like Star Wars is more likely to like The Dark Knight Rises than some arthouse movie.

How do we solve this problem?

Using a second dimension gives us more freedom in organizing movies by similarity



What if we add a second dimension? Perhaps the second dimension is the total number of tickets sold for that movie when it was released in theaters.

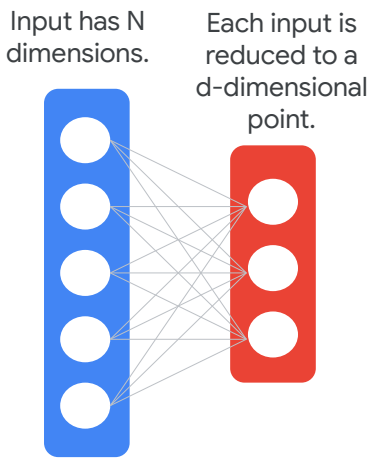
Now, we see that Star Wars and The Dark Knight Rises are close to each other.
Bleu and Memento are close to each other.
Shrek, Incredibles are close to each other as well.
Harry Potter is in between the cartoons and Star Wars in that kids watch it, some adults watch it, and it's a blockbuster.

Notice how adding the second dimension has helped bring movies that are good recommendations closer together. It conforms much better to our intution.

Do we have to stop at two dimensions? Of course not … by adding even more dimensions we can create finer distinctions. And sometimes, these finer distinctions can translate into better recommendations.

Not always … the danger of overfitting exists here too ...

A d-dimensional embedding assumes that user interest in movies can be approximated by d aspects



So, the idea is that we have an input that has N dimensions.

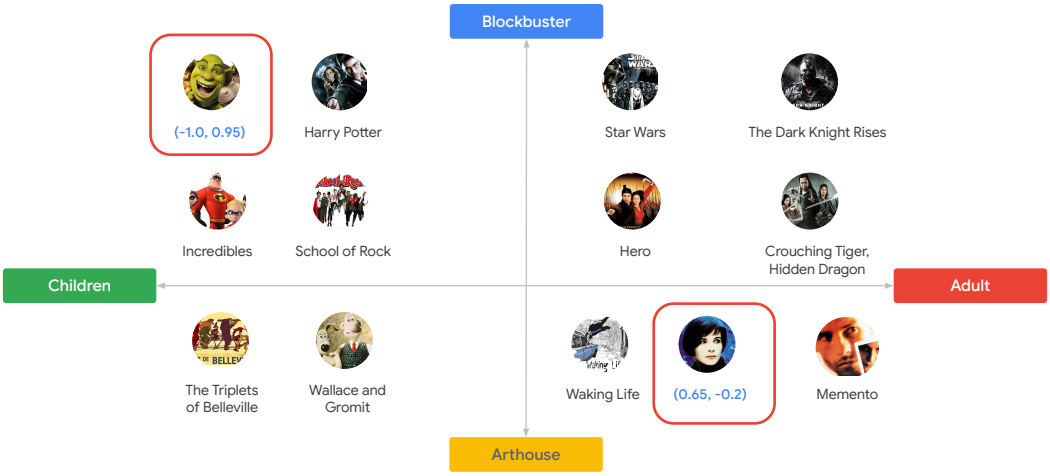
What is N in the case of the movies we looked at?

500,000-right? Remember that the movieId is a categorical feature and we'd normally be one-hot encoding it.
So N = 500,000

On our case, we represented all the movies in a two-dimensional space.
So, d = 2

The key point is that d is much, much less than N.
And the assumption is that user interest in movies can be represented by some d aspects.

The coordinates are called the 2D embedding for the movie



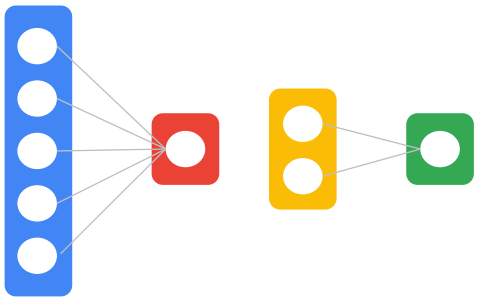
In the previous lesson, we talked about creating embeddings manually, using rules. We used attributes like average age of the viewer and the total ticket sales to take our movies, which would have been in a 500,000-dimensional space and project them into a 2-dimensional space.

In the case of our 2-dimensional embedding, we gave our axes names. Like age and tickets sold. Children vs adult. Arthouse vs blockbuster.

However, it is not essential that these axes have names.

What is important is that we went from 500,000 to 2. Not that we did it by looking at attributes of the movies manually.

It's easier to train a model with d inputs than a model with N inputs



One key reason is this.

Let's say we are training a model to predict whether some user will like a movie.

It is easier to train a model that has d inputs that to train a model that N inputs -- remember that N is much, much larger than d.

The fewer the number of input nodes, the fewer the weights we have to optimize. This means that the model trains faster and has less chance of overfitting.

Embedding is a way of making the problem simpler.

However, we have to do this dimensionality reduction in a way that we don't lose information.
How could we come up with an appropriate embedding?

Embeddings can be learned from data

Example	movieId
0	
1	
2	
3	
4	

#	Shrek	Incredible	Triplets ...	Harry Potter	Star Wars
0	1	0	0	0	0
1	0	1	0	0	0
2	0	0	1	0	0
3	0	0	0	1	0
4	1	0	0	0	0

Embedded Layer

Represent input as sparse vector.

4-dimensional embedding implies we represent movies with 4 latent factors.

You can learn embeddings from the data as part of your normal training process.

No separate training process is needed:

First, take the original input and represent the input as a one-hot encoded array.

Then, send it through an embedding layer

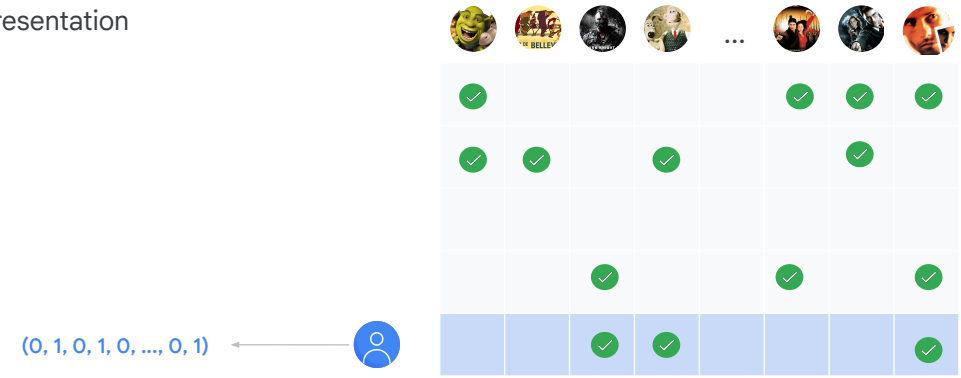
In this approach, the embedding layer is just a hidden layer with one unit per dimension

Because we are training a model with labels, the embeddings get changed based on these labels.
Intuitively the hidden units discover how to organize the items in the d-dimensional space in a way to best optimize the final objective

There is a small problem, though. How much memory is required to store the inputs? You have a categorical input variable with 500,000 possible values, so you have to create 500,000 input nodes and do matrix math of huge matrices ...

Dense representations are inefficient in space and compute

Dense representation



Storing the input vector as a one-hot encoded array is a bad idea.
A dense representation is extremely inefficient, both for storage *and* for compute.
Note that we are calling anything where we store all the values for an input tensor a “dense” tensor.
This has nothing to do with the actual data in the tensor.
Just about how we are storing it.

But consider the data in this matrix. Do you think this matrix is filled densely or sparsely?

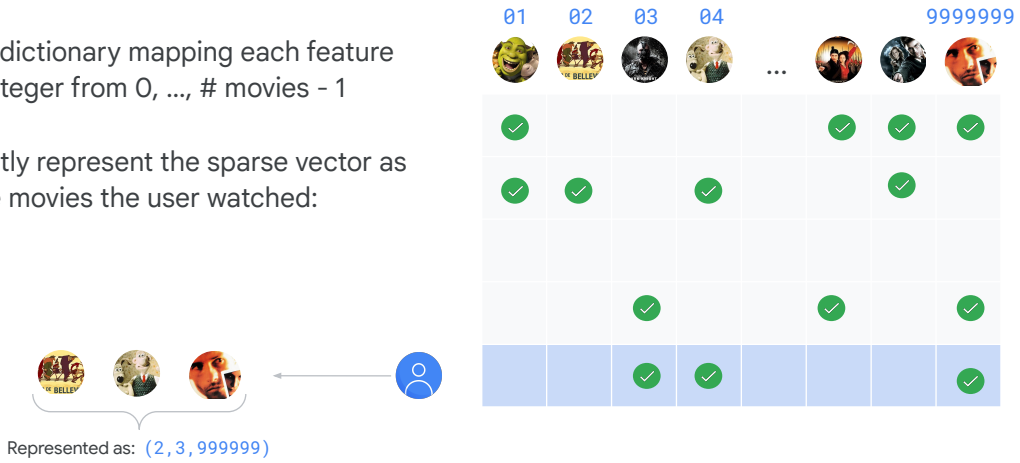
It’s extremely sparse, of course.
Each example (a row in this matrix) represents movies that have been watched by the user.
Think back to your experience. How many movies have *you* rated?

So, we do not want to store the inputs in a dense form. We do not want to store all the values for the tensor.

So, use a sparse representation to hold the example

Build a dictionary mapping each feature to an integer from 0, ..., # movies - 1

Efficiently represent the sparse vector as just the movies the user watched:



So, we do not want to store the inputs in a dense form. We do not want to store all the values for the tensor.
What should we do instead?
It would be good to store the data in a sparse manner, in a compressed way, in memory.
And it would be good to be able to do computations like matrix multiplication directly on the sparse tensors, without having to convert them into dense representations.

The way we do this is to build a dictionary mapping from each feature to an integer. So, Shrek might be the integer zero and Harry Potter might be the integer 300,230. Some arbitrary number. Remember that there is no embedding at this point. At this point, each movie has an arbitrary integer associated with it.

Then, when you a row of the matrix, which represents the movies that a specific user has seen. We simply store the movie-ids for the movies that the user has seen. In the example row, the user has seen three movies, so the sparse tensor has 3 entries in it. Any integer not present in this list, that movie is assumed to not have been watched. So, these 3 entries are 1, and the rest are zero in the equivalent, dense, representation.

So there are two steps here. The pre-processing step computes the dictionary and

then second step uses the dictionary to create an efficient representation.

Representing feature columns as sparse vectors

These are all different ways to create a categorical column.

If you know the keys beforehand:

```
tf.feature_column.categorical_column_with_vocabulary_list('employeeId',
    vocabulary_list = ['8345', '72345', '87654', '98723', '23451']),
```

If your data is already indexed; i.e., has integers in [0-N):

```
tf.feature_column.categorical_column_with_identity('employeeId',
    num_buckets = 5)
```

If you don't have a vocabulary of all possible values:

```
tf.feature_column.categorical_column_with_hash_bucket('employeeId',
    hash_bucket_size = 500)
```


If you are thinking this seems familiar, and just like vocabulary building for categorical columns, you are absolutely correct.
Categorical columns are represented by TensorFlow as sparse tensors.
So, categorical columns are an example of something that is sparse.
TensorFlow can do math operations on sparse tensors without having to convert them into dense.
This saves memory, and optimizes compute.

We looked at how to create a feature cross from categorical columns. That was an example of math that was carried out completely in terms of sparse tensors. This is why, even though we crossed discretized columns of latitude and longitude and then feature crossed the pickup points and dropoff points in our taxifare example, there was no problem with memory or with computation speed.

Code to create an embedded feature column in TensorFlow

Example	movieId
0	
1	
2	
3	
4	

#	Shrek	Incredible	Triplets ...	Harry Potter	Star Wars
0	1	0	0	0	0
1	0	1	0	0	0
2	0	0	1	0	0
3	0	0	0	1	0
4	1	0	0	0	0


```
sparse_movie = layers.sparse_column_with_keys('movieId',

keys=[...])
embedded_movie = layers.embedding_column(sparse_movie,
100)
```


We looked at how to create an embedding column from a feature cross. The same code works for a single categorical column, of course. That's what I'm showing here.

The ability to deal with sparse tensors is why the code to create an embedding column from categorical data in TensorFlow can work without causing memory or speed issues. One of those magic implementation details ...

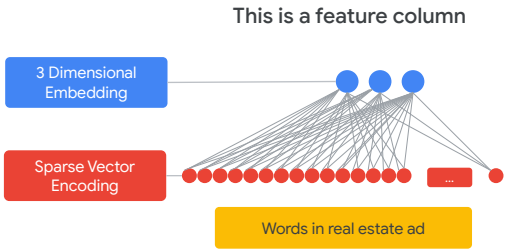
Recall that we said that no separate training process is needed to do embeddings. We just need two steps:
First, take the original input and represent the input.
Second, send it through an embedding layer

The first step is done by taking the input and representing it as a sparse tensor.

The second step is done through the call to embedding_column. But how does that line of code really work?

Embeddings are feature columns that function like layers

```
sparse_word = fc.categorical_column_with_vocabulary_list(
                    'word', vocabulary_list=englishWords)
embedded_word = fc.embedding_column(sparse_word, 3)
```



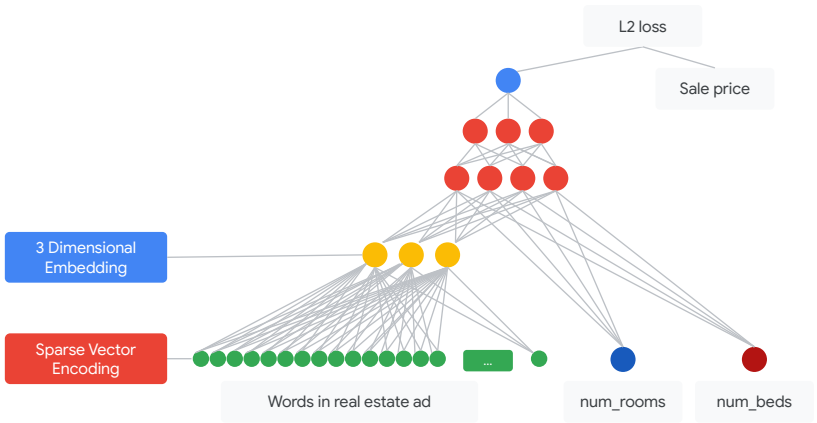
Imagine that you are creating an embedding to represent the key word in a real-estate ad. Let's ignore, for now, how you choose this important word.
Now, words in an ad are natural language and so the potential dictionary is vast. In this case, it could be list of all english words. Tens of thousands of words even if we ignore rare words and scientific jargon.

So, obviously, even though the first layer here takes the word in the real-estate ad and one-hot encodes it, the representation of this in memory will be as a sparse vector. That way, TensorFlow can be efficient in its use of memory.

Once we have the one-hot encoded representation, we pass it through a 3-node layer. This is our embedding, and because we use 3 nodes in that layer, it is a 3-dimensional embedding. Note that even though sparse_word and embedded_word are really feature columns, I am showing them as neural network layers.
That is because, mathematically, they *are* just like neural network layers.

Mathematically, an embedding in this case isn't really different from any other hidden layer in a network. You can view it as a handy adapter that allows the network to incorporate sparse or categorical data well. Key of these slides is to show that you can do this with a regression, classification or ranking problem. Stress that the weights when using a Deep NN are learned by back propagation just as with the other layers.

The weights in the embedding layer are learned through backprop just as with other weights



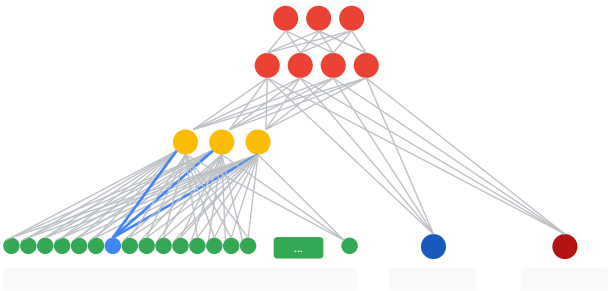
Let's say we use the embedding for the word in a real-estate ad as one of the inputs to a model that predicts the sale price.
We would train such a model based on actual historical sale prices for houses.
In addition to the word in the ad, we might also use number of rooms, number of bedrooms, etc. as inputs.
This is a structured data regression problem, just like the taxifare problem.

Do you see what happens if we try to optimize the weights in all the layers to minimize the error in predicted sale price?
All the weights in all the layers have to be tuned.
The weights get tuned in such a way that the embedding numbers for a word become relevant to its ability to predict sales prices.
Perhaps if the ad includes a word like "view" or "lake", then the sale price has to be higher whereas if the ad includes a word like "foreclosure", the weight has to be lower.
The weights in all the layers will adjust to learn this.

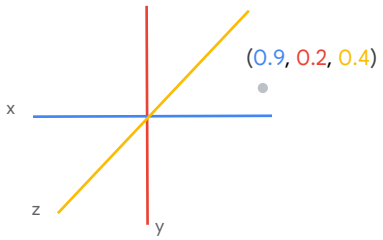
Mathematically, an embedding isn't really different from any other hidden layer in a network. You can view it as a handy adapter that allows the network to incorporate sparse or categorical data well. The weights when using a Deep NN are learned by back propagation just as with the other layers. You can do this with a regression or a classification problem.

Embeddings can be thought of as latent features

Deep Network



Geometric view of a single movie embedding



Now, remember a key fact about the very first layer, the blue layer.
Unlike the yellow nodes, the blue layer is one-hot encoded.
So, if you the word “view”, then only one of those nodes will get turned on.
Let’s say it’s the one in black here. Then, the weights for the links from that black node to the next layer will capture the relevance of the word “view” to this problem.

Therefore, each word is being represented by just three numbers.
Each of the three nodes can be considered as a dimension into which words are being projected.
Edge weights between a movie and hidden layer are coordinate values in this lower-dimensional projection.

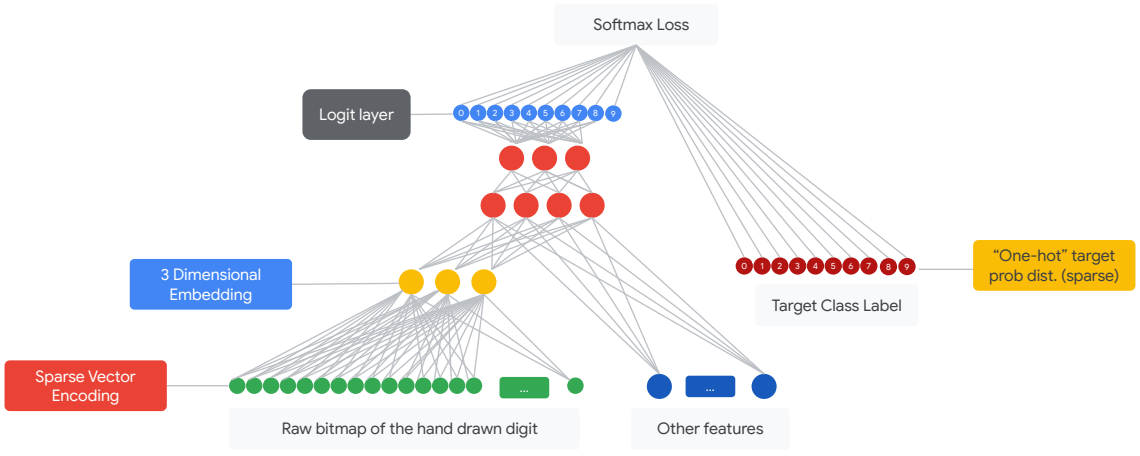
Embeddings provide dimensionality reduction

We started out talking about embeddings for movies, which are categorical features.
Then, we applied the same example to words in an ad -- these are text features.
But embeddings are not just for categorical features.

Here, I'm showing you a classic machine learning problem called MNIST.
The idea is to recognize handwritten digits from scanned images.
So you take each image, and each of the pixels in the image is an input.
That's what I mean by "raw bitmap" here. Now, the images are 28x28 so there are 784 pixels in the bitmap.
Now, consider this array of 784 numbers. Most of the array corresponds to blank pixels ...

Embeddings are useful here also.

Embeddings provide dimensionality reduction

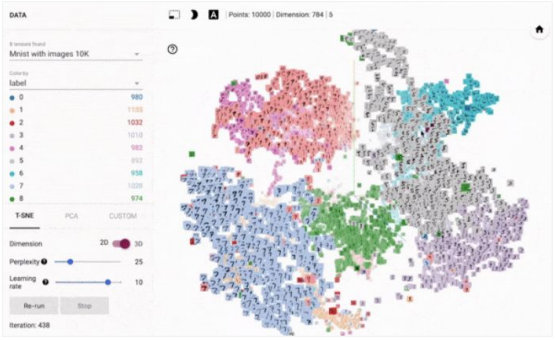


We take the 784 number, represent them as a sparse tensor -- essentially, we only save the pixels where the hand-written digit appears --
Then we pass it through a 3D embedding. We can then have a normal 2-layer neural network.
We could pass in other features if we wanted.
Then, we train the model to predict the actual number in the image based on labels.

Why do I have a logit layer here? These from the output layer of
A logit is what the output has to be for a classification problem.
When we use a LinearClassifier or a DNNClassifier, the output layer is a logit. One single logit.
But that is if you have only one output. In the case of the MNIST problem, we have 10 total classes -- essentially the digits 0, 1, 2, up till 9.
That's why I have a logit layer, one logit for each of the possible digits.
When we have a logit layer, as opposed to a single logit, there is no guarantee that the total probability of all the digits will equal one.
That's the role of the softmax. It normalizes the individual logits.

Sorry for the digression. We were talking about embeddings.
So, here, when we train the model to recognize handwritten digits, each image will be represented by 3 numbers.
Unlike in the categorical case, though, the raw bitmap is not one-hot encoded, so we won't get 3 numbers for each pixel,
Instead, the 3 numbers correspond to all the pixels that are turned on for each image.

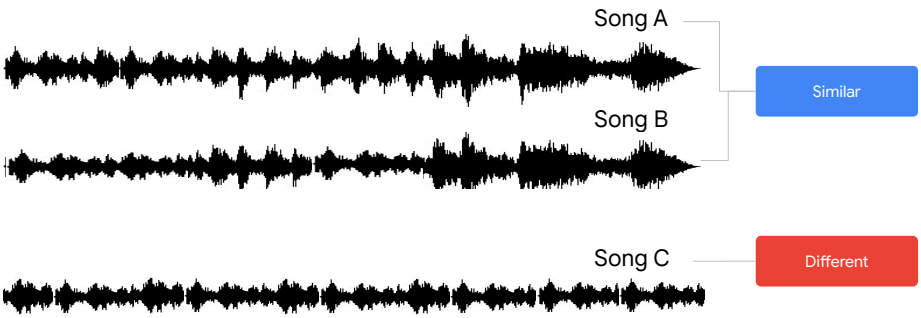
The result of embedding is such that similar items are close to each other



In TensorBoard, you can visualize these embeddings, the 3D vector that corresponds to each 784-pixel image.
Here, we have assigned different colors to the labels.
As you can see, something cool happens.
All the 5s cluster together in 3D space.
As do all the 7s, and all the zeros.
In other words, the 3D numbers that represent each handwritten image are such that similar items are close to each other in the 3D space.

This is true of embeddings for categorical variables, for natural language text, and for raw bitmaps.
If you take a sparse vector encoding and pass it through a embedding column and then use that embedding column as the input to a DNN and train the DNN, then the trained emebddings will have this similarity property. As long as, of course, you have enough data and your training achieved good accuracy.

You can take advantage of this similarity property of embeddings



You can take advantage of this similarity property in other situations

Suppose, for example, your task is to find a song similar to this song, what you could do is to create an embedding of the audio associated with songs.
Essentially, you take the audio clip and represent it as an array of values. Then, just like with the MNIST image, you take the array and pass it through an embedding layer. You use this to train some reasonable machine learning problem. Perhaps you use the audio signal to train a model to predict the musical genre or the next musical note. Regardless of what you train the model to predict, the embedding will give you a lower-dimensional representation of the audio clip.
Now, you can simply compute the Euclidean distance between clips as a measure of the similarity of two songs.

You could also use the embedding vectors as inputs to a clustering algorithm.

The similarity idea can also be used to jointly embed diverse features (e.g. text in two different languages, text and audio, etc.) to define similarity between them

A good starting point for number of embedding dimensions

Lower dimensions
less accuracy, more lossy compression.

Higher dimensions
overfitting, slow training.

$$\text{dimensions} \approx \sqrt[4]{\text{possible values}}$$

Empirical tradeoff.

In all our examples, we used 3 for the number of embeddings. You can use different numbers, of course.

But what number should you use?

This is a hyperparameter to your machine learning model. You will have to try different numbers of embedding dimensions because there is a tradeoff.
Higher-dimensional embeddings can more accurately represent the relationships between input values.
However, the more dimensions you have, the greater the chance of overfitting.
Also, the model gets larger and leads to slower training

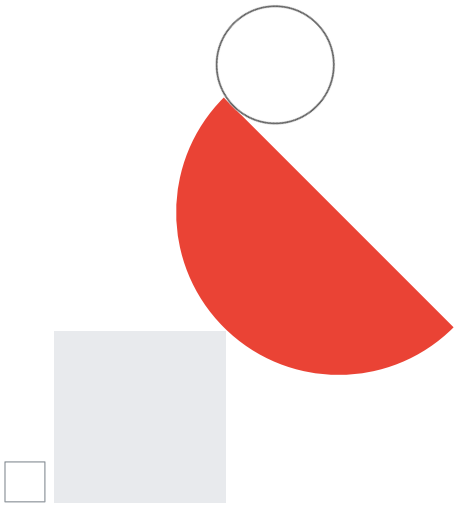
A good starting point is to go with the fourth root of the total number of possible values.

For example, if you are embedding movieId and you have 500,000 movies in your catalog, a good starting point might be the fourth root of 500,000.
The square root of 500,00 is about 700. And the square root of 700 is about 26. So, I'd probably start at a round 25.

In the hyperparameter tuning, I would specify a search space of perhaps 15 to 35.

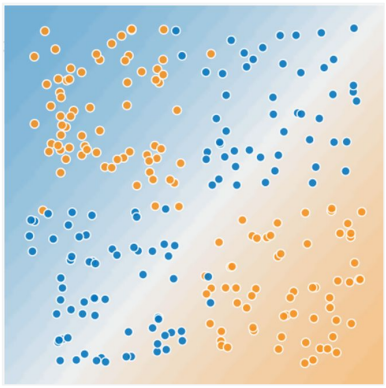
But this is only a rule of thumb, of course.

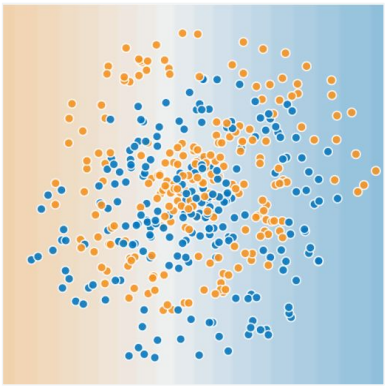
The Science of Neural Networks



Welcome to The Science of Neural Networks.
Now that we have seen the art of machine learning, in this module we will now be diving deep into the science, specifically with neural networks.

Feature crosses help linear models work in nonlinear problems, but there tends to be a limit ...





We recently saw that feature crosses did very well on a problem like this. If x1 is the horizontal dimension and x2 is the vertical dimension there was no linear combination of the two features to describe this distribution. It wasn't until we did some feature engineering and crossed x1 with x2 to get a new feature, x3 which equals x1 times x2, were we able to describe our data distribution. So manual, handcrafted feature engineering can easily solve all of our non-linear problems, right?...

Unfortunately, the real world almost never has such easily described distributions so feature engineering even after years of the brightest people working on it can only get so far.
For instance, what feature crosses would you need to model this distribution? It looks sort of like two circles over top of each other or maybe two spirals, but whatever it is it is very messy.
This kind of example sets up the usefulness of neural networks since they can algorithmically create very complex feature crosses and transformations. We can imagine much more complicated spaces than even this spiral that really necessitate the use of neural networks.

Combine features as an alternative to feature crossing

Structure the model so that features are combined.
Then the combinations may be combined.

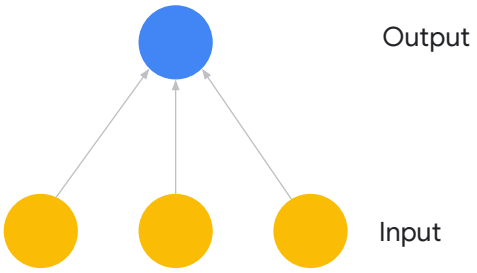
How to choose the combinations?
Get the model to learn them.

Neural networks can help as an alternative to feature crossing by combining features. When we are designing our neural network architecture, we want to structure the model in such a way that are features are combined. Then we want to add another layer to combine our combinations and then another layer to combine those combinations, and so on.

How do we choose the right combinations of our features and the combinations of those, etc? You get the model to learn them through training of course!

This is the basic intuition behind neural nets. This approach isn't necessarily better than feature crosses, but it is a flexible alternative that works well in many cases.

A Linear Model can be represented as nodes and edges



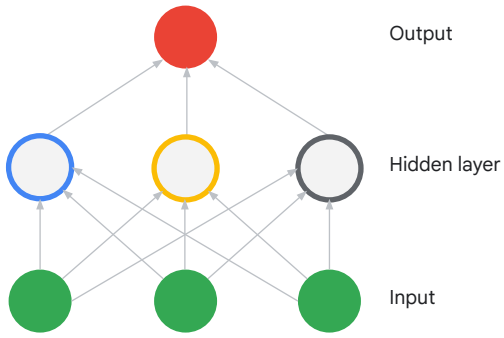
Here is a graphical representation of a linear model. We have three inputs, x1, x2, and x3, shown by the blue circles. They're combined with some weight, given on each edge, to produce an output. There often is an extra bias term, but for simplicity it isn't shown here. This is a linear model since it is of the form y = w1 * x1 + w2 * x2 + w3 * x3.

Add Complexity: Non-Linear?

$h_1 = w_1 * x_1 + w_4 * x_2 + w_7 * x_3$

$h_2 = w_2 * x_1 + w_5 * x_2 + w_8 * x_3$

$h_3 = w_3 * x_1 + w_6 * x_2 + w_9 * x_3$



Now let's add a hidden layer to our network of nodes and edges. Our input layer has three nodes and our hidden layer also has three, but now hidden, nodes. Since this is a completely connected layer there are 3 * 3 edges or 9 weights. Surely this is a non-linear model now that we can use to solve our non-linear problems right? Unfortunately not, let's break it down.

The input to the first hidden node is the weighted sum w1 * x1 + w4 * x2 + w7 * x3.

The input to the second hidden node is the weighted sum w2 * x1 + w5 * x2 + w8 * x3.

The input to the third hidden node is the weighted sum w3 * x1 + w6 * x2 + w9 * x3

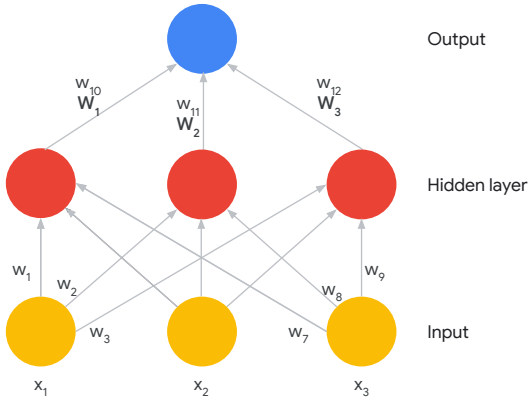
Add Complexity: Non-Linear?

$$\text{Output} = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

$$((w_{10} * w_1) + (w_{10} * w_4) + (w_{10} * w_7)) * x1$$
$$+ ((w_{11} * w_1) + (w_{11} * w_4) + (w_{11} * w_7)) * x2$$
$$+ ((w_{12} * w_1) + (w_{12} * w_4) + (w_{12} * w_7)) *$$

x3

$$= W_1 * x_1 + W_2 * x_2 + W_3 * x_3$$



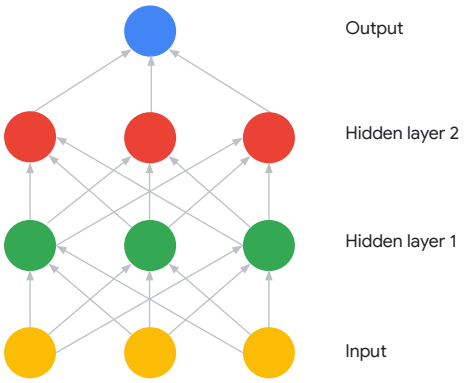
Combining it all together, at the output node we have, w10 * h1 + w11 * h2 + w12 * h3. Remember though that h1, h2, and h3 are just linear combinations of the input features.

Therefore, expanding it out we're left with a complex set of weight constants multiplied by each input value, x1, x2, and x3.

\\=(w_{10}*w_1+w_{11}*w_2+w_{12}*w_3)*x_1\\+(w_{10}*w_4+w_{11}*w_5+w_{12}*w_6)*x_2\\+(w_{10}*w_7+w_{11}*w_8+w_{12}*w_9)*x_3

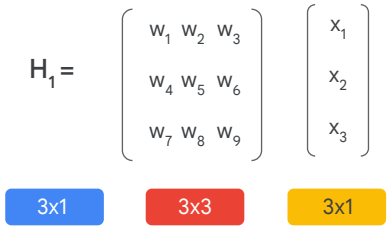
We can substitute each group of weights for a new weight. Look familiar? This is exactly the same linear model as before despite adding a hidden layer of neurons. So what happened?

Add Complexity:
Non-Linear?



What if we added another hidden layer? Unfortunately, this once again collapses all of the way back down into a single weight matrix multiplied by each of the three inputs. It is the same linear model! We could continue this process ad infinitum and it would be the same result, albeit a lot more costly computationally for training or prediction from a much, much more complicated architecture than needed.

More Complex: Non-Linear?

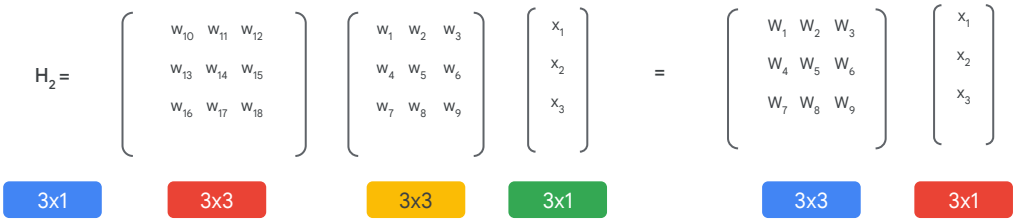


Thinking about this from a linear algebra perspective, you are multiplying multiple matrices together in a chain.

In this small example, I first multiply a 3x3 matrix, the transpose of the weight matrix between the input layer and hidden layer 1, by my 3x1 input vector resulting in a 3x1 vector, which are the values at each hidden neuron in hidden layer one.

H_1=\begin{bmatrix} w_1 & w_2 & w_3\\ w_4 & w_5 & w_6\\ w_7 & w_8 & w_9\end{bmatrix}
\begin{bmatrix} x_1\\ x_2\\ x_3\end{bmatrix}

More Complex: Non-Linear?



To find the second hidden layer's neurons' values, I multiply the transpose of its 3x3 weight matrix that connects hidden layer one with hidden layer two to my resultant vector at hidden layer 1. As you can see the two 3x3 weight matrices can be combined into one 3x3 matrix by first calculating the matrix product from the left instead of from the right. This still gives the same shape for H2, the second hidden layer neurons' value vector.

H_2=\begin{bmatrix} w_{10} & w_{11} & w_{12}\\ w_{13} & w_{14} & w_{15}\\ w_{16} & w_{17} & w_{18}\end{bmatrix}
\begin{bmatrix} w_1 & w_2 & w_3\\ w_4 & w_5 & w_6\\ w_7 & w_8 & w_9\end{bmatrix}
\begin{bmatrix} x_1\\ x_2\\ x_3\end{bmatrix}

=\begin{bmatrix} W_1 & W_2 & W_3\\ W_4 & W_5 & W_6\\ W_7 & W_8 & W_9\end{bmatrix}
\begin{bmatrix} x_1\\ x_2\\ x_3\end{bmatrix}

More Complex: Non-Linear?

Adding in the final layer between hidden layer 2 and the output layer I need to multiply the preceding steps by the transpose of the weight matrix between the last two layers. Even though when feeding forward through a neural network you perform the matrix multiplication from right to left, by applying it from left to right you can see that our large chain of matrix multiplications collapses down into just a 3 valued vector.

If you train this model and just the simple linear regression case of 3 weights side by side and they both fall into the same minimum of the loss surface, then even though I did a ton of computation to calculate all 21 weights in my matrix product chain, when condensed down into the lower equation the weight will exactly match the training simple linear regression's weights. All of that work for the same result!

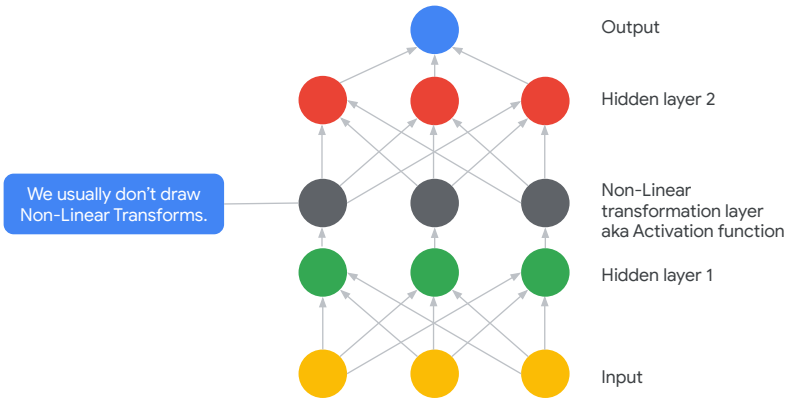
You're probably thinking now, "Hey, I thought neural networks were all about adding layers upon layers of neurons. How can I do deep learning when all of the layers collapse into just one?" I've got good news for you, there is an easy solution!

\hat y=\begin{bmatrix}w_{19} & w_{20} & w_{21}\end{bmatrix}
\begin{bmatrix} w_{10} & w_{11} & w_{12}\\ w_{13} & w_{14} & w_{15}\\ w_{16} & w_{17} & w_{18}\end{bmatrix}
\begin{bmatrix} w_1 & w_2 & w_3\\ w_4 & w_5 & w_6\\ w_7 & w_8 & w_9\end{bmatrix}
\begin{bmatrix} x_1\\ x_2\\ x_3\end{bmatrix}

\hat y=\begin{bmatrix}W'_1 & W'_2 & W'_3\end{bmatrix}

$$\begin{bmatrix} x_1\\ x_2\\ x_3\end{bmatrix}$$

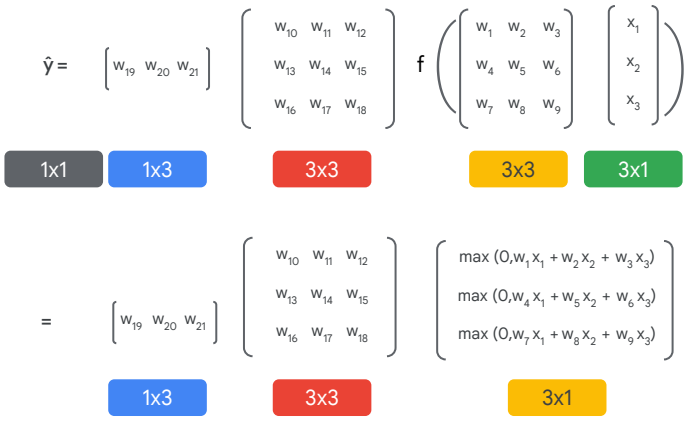
Adding a Non-Linearity



The solution is adding a non-linear transformation layer which is facilitated by a non-linear activation function such as Sigmoid, Tanh, or ReLU. In thinking of terms of the graph such as we make in TensorFlow, you can imagine each neuron actually having two nodes. The first node being the result of the weighted sum W * x + b and the second node being the result of that being passed through the activation function. In other words, there are the inputs to the activation function followed by the outputs of the activation function so the activation function acts as the transition point between.

Adding in this non-linear transformation is the only way to stop the neural network from condensing back into a shallow network. Even if you have a layer with non-linear activation functions in your network, if elsewhere in the network you have 2 or more layers with linear activation functions, those can still be collapsed into just one network. Usually neural networks have all layers non-linear for the first n minus 1 layers and then have the final layer transformation be linear for regression or sigmoid or softmax, which we will talk about soon, for classification. It all depends on what you want the output to be.

Adding a Non-Linearity



Thinking about this again from a linear algebra perspective, when we apply a linear transformation to a matrix or vector we are multiplying a matrix or vector to it leading to our desired shape and result. Such as when I want to scale a matrix, I can multiply it by a constant, but truly what you are doing is multiplying it by an identity matrix multiplied by that constant, so it is a diagonal matrix with that constant all along the diagonal. This can still be collapsed into just a matrix product.

However, if I add a non-linearity, what I am doing is not able to be represented by a matrix, since I am element-wise applying a function to my input. For instance, if I have a non-linear activation function between my first and second hidden layers, I am applying a function to the product of the transpose of my first hidden layer's weight matrix and my input vector. The lower equation is if my activation function in a ReLU.

Since I cannot represent this transformation in terms of linear algebra, I can no longer collapse that portion of my transformation chain, thus the complexity of my model remains and doesn't collapse into just a linear combination of the inputs.

Note, that I can still collapse the second hidden layer weight matrix and the output layer weight matrix since there is no non-linear function being applied there. This means that whenever there are 2 or more linear layers consecutively, they can always be collapsed back into one layer no matter how many there are. Therefore, to have the most complex functions being created by your network, it's best to have your entire network have linear activation functions except at the last layer in case you might need a certain type of output at the end.

$$\hat y=\begin{bmatrix}w_{19} & w_{20} & w_{21}\end{bmatrix}
\begin{bmatrix} w_{10} & w_{11} & w_{12}\\ w_{13} & w_{14} & w_{15}\\ w_{16} & w_{17} & w_{18}\end{bmatrix}
f\left(\begin{bmatrix} w_1 & w_2 & w_3\\ w_4 & w_5 & w_6\\ w_7 & w_8 & w_9\end{bmatrix}
\begin{bmatrix} x_1\\ x_2\\ x_3\end{bmatrix}\right)$$

$$=\begin{bmatrix}w_{19} & w_{20} & w_{21}\end{bmatrix}
\begin{bmatrix} w_{10} & w_{11} & w_{12}\\ w_{13} & w_{14} & w_{15}\\ w_{16} & w_{17} & w_{18}\end{bmatrix}
\begin{bmatrix} max(0, w_1 x_1 + w_2 x_2 + w_3 x_3)\\ max(0, w_4 x_1 + w_5 x_2 + w_6 x_3)\\ max(0, w_7 x_1 + w_8 x_2 + w_9 x_3)\end{bmatrix}$$

Quiz: Non-linearity

Why is it important adding non-linear activation functions to neural networks?

A. Adds regularization

B. Increases the number of dimensions

C. Invokes early stopping

D. Stops the layers from collapsing back into just a linear model

Question: Why is it important adding non-linear activation functions to neural networks?

Quiz: Non-linearity

Why is it important adding non-linear activation functions to neural networks?

A. Adds regularization

B. Increases the number of dimensions

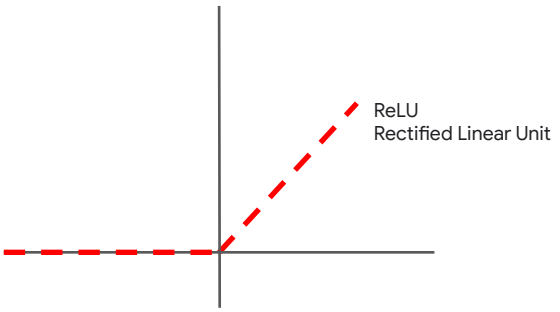
C. Invokes early stopping

D. Stops the layers from collapsing back into just a linear model

Answer: The correct answer is because it stops the layers from collapsing back into just a linear model. Not only do non-linear activation functions help create interesting transformations to our data's feature space, but it allows for deep composition of functions. As we explained, if there are any 2 or more layers with linear activation functions, this product of matrices can be summarized by just one matrix times the input feature vector. Therefore, you end up with slower model with more computation, but with all of your functional complexity reduced.

Non-linearities do not add regularization to the loss function and they do not invoke early stopping. Even though, non-linear activation functions, do create complex transformations of the vector space, the dimension does not change and you remain in the same vector space, albeit stretched, squished, or rotated.

Our favorite non-linearity is the Rectified Linear Unit



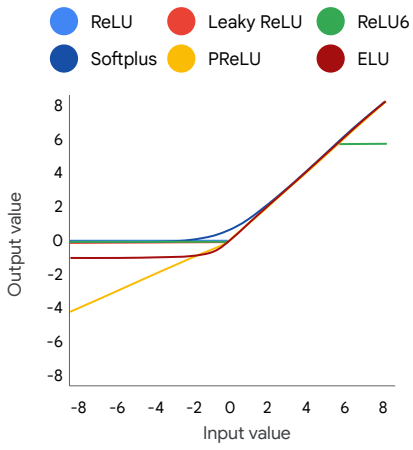
$f(x) = \max(0, x)$

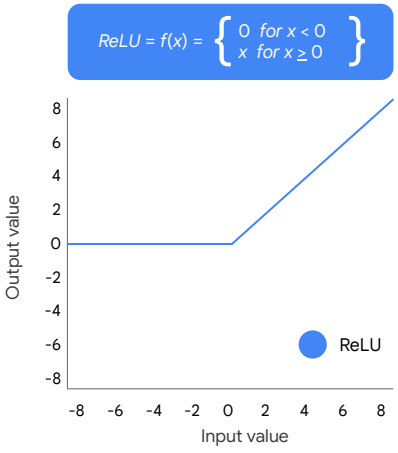
As mentioned in one of our previous courses, there are many non-linear activation functions with sigmoid, and the scaled and shifted sigmoid, hyberbolic tangent being some of the earliest. However, as mentioned before these can have saturation which leads to the vanishing gradient problem where with zero gradients the model's weights don't update and training halts.

The Rectified Linear Unit, or ReLU for short, is one of our favorites because it's simple and works well. In the positive domain it is linear so we don't have saturation whereas in the negative domain the function is 0. Networks with ReLU hidden activations often have 10 times the speed of training than networks with sigmoid hidden activations. However, due to the negative domain's function always being zero, we can end up with ReLU layers dying. What I mean by this is that when you start getting inputs in the negative domain then the output of the activation will be zero which doesn't help in the next layer in getting the inputs into the positive domain. This compounds and creates a lot of zero activations. During backpropagation when updating the weights, since we have to multiply our error's derivative by the activation, we end up with a gradient of zero, thus a weight update of 0, and thus the weights don't change and training fails for that layer.

Fortunately, a lot of clever methods have been developed to slightly modify the ReLU to ensure training doesn't stall, but still with much of the benefits of the vanilla ReLU.

There are many different ReLU variants

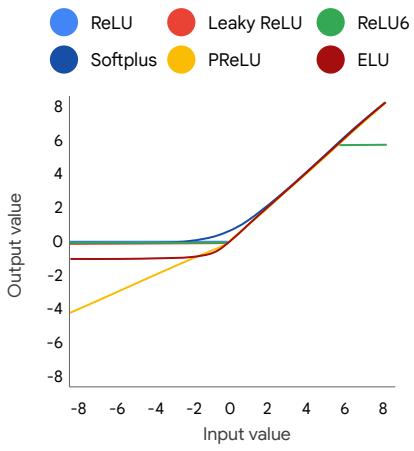


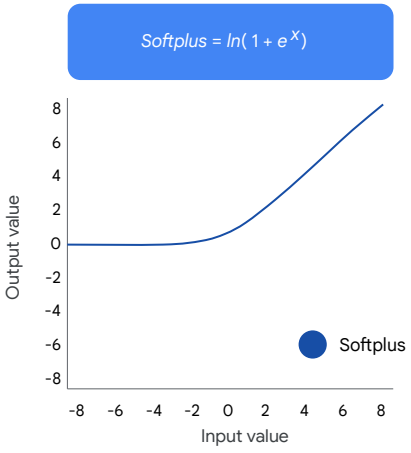


Here again is the vanilla ReLU. The maximum operator can also be represented by the piecewise linear equation where less than 0 the function is 0 and greater than or equal to 0 the function is x.

ReLU = f(x)=\begin{Bmatrix}
0 \ \ \ for \ x < 0
\\
x \ \ \ for \ x \geq 0
\end{Bmatrix}

There are many different ReLU variants

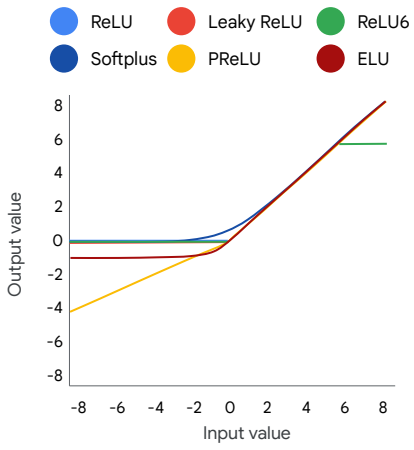


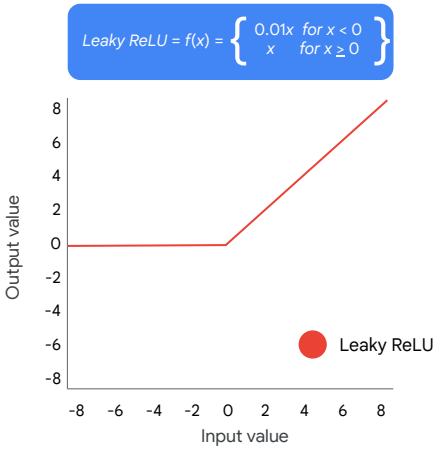


A smooth approximation of the ReLU's function is the analytic function of the natural log of 1 plus the exponential of x. This is called the softplus function. Interestingly, the derivative of the the softplus function is the logistic function. The pros of using the softplus function are it is continuous and differentiable at 0, unlike the ReLU function. However, due to the natural log and exponential, there is added computation compared to ReLUs and ReLUs still have just as good of results in practice. Therefore, softplus is usually discouraged to be used in deep learning.

Softplus=ln(1+e^x)

There are many different ReLU variants



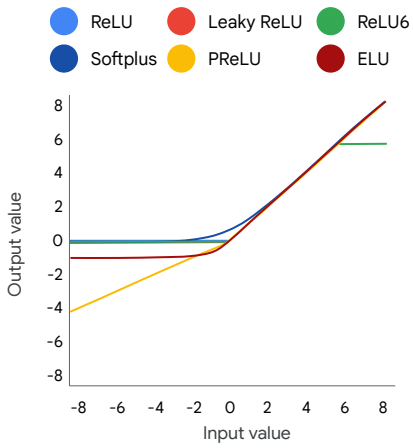


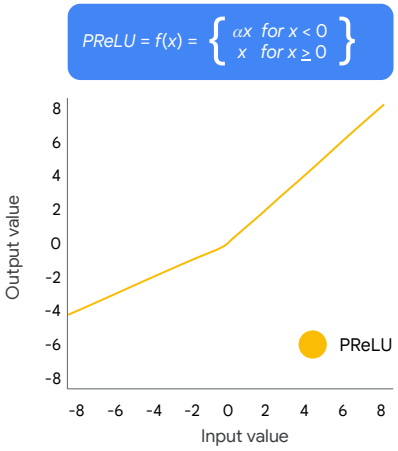
To try and solve our issue of dying ReLUs, due to zero activations, the leaky ReLU was developed. Just like ReLUs, leaky ReLUs have a piecewise linear function, however in the negative domain rather than 0, they have a nonzero slope, specifically 0.01.

This way when the unit is not activated, leaky ReLUs still allow a small non-zero gradient to pass through them, which hopefully will allow weight updating and training to continue.

$$Leaky\ ReLU = f(x)=\begin{Bmatrix} 0.01x \ \ \ for \ x < 0 \\ x \ \ \ for \ x \geq 0 \end{Bmatrix}$$

There are many different ReLU variants



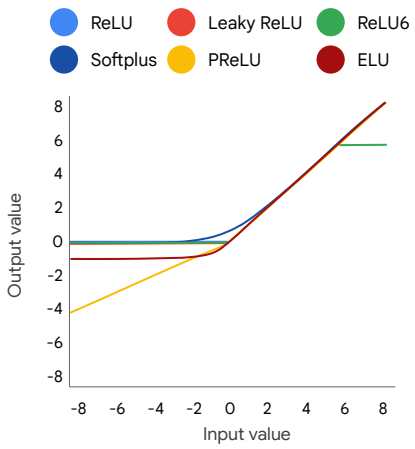


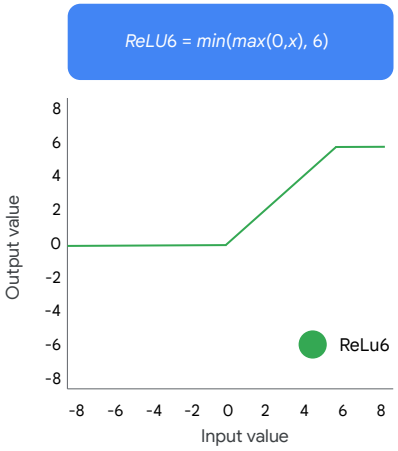
Taking this leaking idea one step further, is the parametric relu, or PReLU for short. Here rather than arbitrarily allowing one one-hundreth of x through in the negative domain, it let's alpha of x through. But what is the parameter alpha supposed to be? In the graph, I set alpha to be 0.5 for visualization purposes, but in practice it is actually a learned parameter from training along with the other neural network parameters. This way rather than us setting its value, the value will be determined during training via the data and should learn a more optimal value than we a priori could set. Notice that when alpha is less than 1, the formula can be rewritten back into the compact form using the maximum, specifically the max of x or alpha times x.

There are also randomized leaky ReLUs where instead of alpha being trained, it is sampled from a uniform distribution randomly. This can have an effect similar to dropout since you technically have a different network for each value of alpha and therefore it is making something similar to an ensemble. At test time, all of the values of alpha are averaged together to get a deterministic value to use for predictions.

PReLU = f(x)=\begin{Bmatrix}
\alpha x \ \ \ for \ x < 0
\\
x \ \ \ for \ x \geq 0
\end{Bmatrix}

There are many different ReLU variants





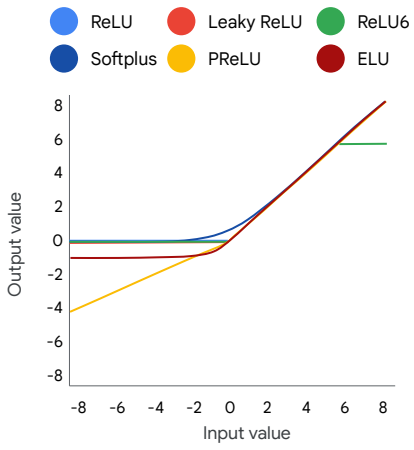
There is also the ReLU6 variant. This is another piecewise linear function with three segments. Like a normal ReLU it is 0 in the negative domain, however in the positive domain, the ReLU6 function is capped at 6. You're probably thinking, why is it capped at 6?

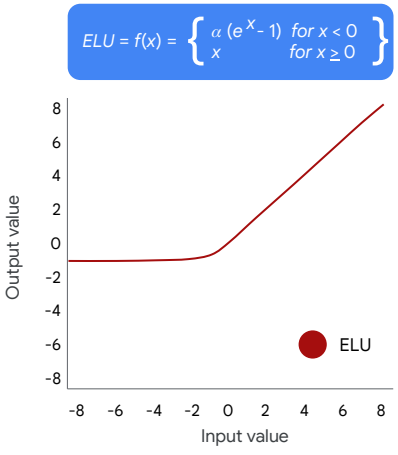
You can imagine one of these ReLU units having only 6 replicated bias-shifted Bernoulli units, rather than an infinite amount due to the hard cap. In general, these are called ReLU-n units where n is the cap value. In testing, 6 was found to be the most optimal value.

ReLU6 units can help models learn sparse features sooner. They were first used in convolutional deep belief networks on the CIFAR-10 image dataset. They also have the useful property of preparing the network for fixed point precision for inference. If the upper limit is unbounded, then you lose too many bits to the Q part of a fixed-point number whereas with a upper limit of 6, it leaves enough bits to the fractional part of the number making it represented well enough to do good inference

ReLU6 = min(max(0, x), 6)

There are many different ReLU variants

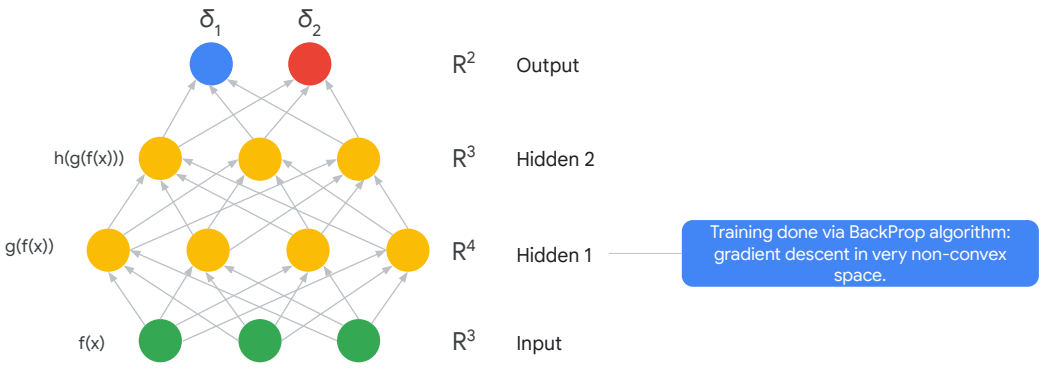




Lastly, there is the Exponential Linear Unit or ELU. It is approximately linear in the non-negative portion of the input space and is smooth, monotonic, and most importantly non-zero in the negative portion of the input space. They are also better zero centered than vanilla ReLUs which can speed up learning. The main drawback of ELUs are that they are more computationally expensive than ReLUs due to having to calculate the exponential.

ELU = f(x)=\begin{Bmatrix}
\alpha (e^{x}-1) \ \ \ for \ x < 0
\\
x \ \ \ \ \ \ \ \ \ \ \ \ \ for \ x \geq 0
\end{Bmatrix}

Neural Nets can be arbitrarily complex



Neural networks can be arbitrarily complex. There can be many layers, neurons per layer, outputs, inputs, different types of activation functions, etc.

What is the purpose of multiple layers? Each layer I add, adds to the complexity of functions I can create. Each subsequent layer is a composition of the previous functions. Since we are using non-linear activation functions in my hidden layers, I am creating a stack of nested transformations that rotate, stretch, and squeeze my data. Remember, the purpose of doing all of this is to transform my data in such a way that I can nicely fit a hyperplane to it for regression or separate my data with hyperplanes for classification. We are mapping from the original feature space to some new convoluted feature space.

What does adding additional neurons to a layer do? Each neuron I add, adds a new dimension to my vector space. If I begin with three input neurons I start in R3 vector space, but if my next layer has four neurons then I move to an R4 vector space. Back when we talked about kernel methods in a previous course, we had a dataset that couldn't be easily separated with a hyperplane in the original input vector space, but by adding a dimension and then transforming the data to fill that new dimension in just the right way, we were then easily able to make a clean slice between the two classes of data. The same applies here with neural networks.

What might having multiple output nodes do?
Having multiple output nodes allows you to compare to multiple labels and then propagate the corresponding errors backwards. You can imagine doing image

classification where there are multiple entities or classes within each image. We can’t just predict one class because there may be many, so having this flexibility is great.

Quiz: Neural Network Complexity

Neural networks can be arbitrarily complex.
To increase hidden dimensions, I can add _____.
To increase function composition, I can add _____.
If I have multiple labels per example, I can add _____.

A. Layers, neurons, outputs

B. Neurons, layers, outputs

C. Layers, outputs, neurons

D. Neurons, outputs, layers

Question: Neural networks can be arbitrarily complex. To increase hidden dimensions, I can add BLANK. To increase function composition, I can add BLANK. If I have multiple labels per example, I can add BLANK.

Quiz: Neural Network Complexity

Neural networks can be arbitrarily complex.
To increase hidden dimensions, I can add _____.
To increase function composition, I can add _____.
If I have multiple labels per example, I can add _____.

A. Layers, neurons, outputs

B. Neurons, layers, outputs

C. Layers, outputs, neurons

D. Neurons, outputs, layers

Answer: The correct answer is neurons, layers, outputs. To change hidden dimensions, I can change a layer's number of neurons since that determines the dimension of the vector space my intermediate vector is in. If a layer has 4 neurons then it is in R4 vector space and if a layer has 500 neurons it is in R500 vector space meaning it has 500 real dimensions! Adding a layer doesn't change the dimension of the previous layer and it might not even change the dimension in its layer unless it has a different number of neurons than the previous layer.

What additional layers do add is a greater composition of functions. Remember, g(f(x)) is the composition of the function g with the function f on the input x. Therefore, I first transform x by f and then transform that result by g. The more layers I have the deeper the nested functions go. This is great for combining non-linear functions together to make very convoluted feature maps that are hard for humans to construct but great for computers and allow us to better get our data into a shape that we can learn and gain insights from.

Speaking of insights, we receive those through our output layers, where during inference, those will be the answers to our ML formulated problem. If you only want to know the probability of an image being a dog, then you can get by with only one output node, but if you wanted to know the probability of an image being a cat, dog, bird, or moose, then you would need to have a node for each one.

The other 3 answers are all wrong since they get two or more of the words wrong.

DNNRegressor usage is similar to LinearRegressor

```
myopt = tf.train.AdamOptimizer(learning_rate=0.01)

model = tf.estimator.DNNRegressor(model_dir=outdir,
                    hidden_units=[100, 50, 20],
                                  feature_columns=INPUT_COLS,
                                  optimizer=myopt,
                                  dropout=0.1)

NSTEPS = (100 * len(traindf)) / BATCH_SIZE
model.train(input_fn=train_input_fn, steps=NSTEPS)
```

- Use momentum-based optimizers e.g. Adagrad (the default) or Adam.
- Specify number of hidden nodes.
- Optionally, can also regularize using dropout.

In TensorFlow, using the Estimator API, using a DNN Regressor is very similar to using a Linear Regressor with only a few parameters to the code that need to be added. We can use momentum-based optimizers such as the default Adagrad or we can try many others such as Adam. Also, we now have to add a parameter named hidden_units which is a list. The number of items in this list is the number of hidden layers and the values of each list item is the number of neurons for that particular hidden layer. You will also notice a new parameter named dropout. We'll cover this more in a few minutes, but for now this is used to turn individual neurons on and off for each example in hopes of having better generalization performance. Please look at the TensorFlow documentation for the complete set of parameters you can configure.

These are all things that could be hyperparameterized so that you can tune your model to have the best generalization performance.

Three common failure modes for gradient descent

Problem	Gradients can vanish	Gradients can explode	ReLu layers can die
Insight	Each additional layer can successively reduce signal vs. noise	Learning rates are important here	Monitor fraction of zero weights in TensorBoard
Solution	Using ReLu instead of sigmoid/tanh can help	Batch normalization (useful knob) can help	Lower your learning rates

Backpropagation is one of the traditional topics in an ML/neural networks course, but at some level it's kind of like teaching people how to build a compiler. It's essential for deeper understanding, but not necessarily needed for initial understanding. The main thing to know is that there's an efficient algorithm for calculating derivatives and TensorFlow will do it for you automatically. There are some interesting failure cases to talk about though such as vanishing gradients, exploding gradients, and dead layers.

First, during the training process, especially for deep networks, gradients can vanish. Each additional layer in your network can successively reduce signal vs. noise. An example of this is when using sigmoid or tanh activation functions throughout your hidden layers. As you begin to saturate, you end up in the asymptotic regions of the function which begin to plateau. The slope is getting closer and closer to approximately zero. When you go backwards through the network during backprop, your gradient can become smaller and smaller because you are compounding all of these small gradients until the gradient completely vanishes. When this happens, your weights are no longer updating and therefore training grinds to a halt. A simple way to fix this is to use non-saturating, non-linear activation functions such as ReLUs, ELUs, etc.

Next, we can also have the opposite problem, where gradients explode, by getting bigger and bigger until our weights get so large we overflow. Even starting with relatively small gradients such as a value of 2 can compound and become quite large over many layers. This is especially true for sequence models with long sequence lengths. Learning rates can be a factor here because in our weight updates remember we multiply the gradient with the learning rate and then subtract that from the current weight. So even if the gradient isn't that big, with a learning rate greater than 1 it can now become too big and cause problems for us and our network.

There are many techniques to try and minimize this such as weight regularization and smaller batch sizes. Another technique is gradient clipping, where we check to see if the norm of the gradient exceeds some threshold, which you can hyperparameter tune, and if so then you can rescale the gradient components to be below your maximum.

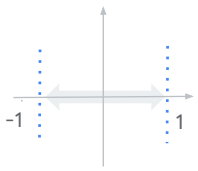
Another useful technique is batch normalization, which solves a problem called internal covariate shift. It speeds up training because gradients flow better. It also can often use a higher learning rate and might be able to get rid of dropout which slows computation down due to its own kind of regularization due to mini-batch noise. To perform batch normalization, first find the mini-batch mean, then the mini-batches standard deviation, then normalize the inputs to that node, then scale and shift by y = gamma times x plus beta where gamma and beta are learned parameters. If gamma = sqrt(var(x)) & beta=mean(x), the original activation is restored. This way you can control the range of your inputs so that they don't become too large. Ideally you would like to keep your gradients as close to one as possible, especially for very deep nets, so that you don't compound and eventually underflow or overflow.

Another common failure mode of gradient descent is that ReLU layers can die. Fortunately, using TensorBoard we can monitor the summaries during and after training of our neural network models. If using a canned DNN estimator, there is automatically a scalar summary saved for each DNN hidden layer showing the fraction of zero values of the activations for that layer. ReLUs stop working when their inputs keep them in the negative domain giving their activation a value of zero. It doesn't end there because then their contribution to the next layer is 0 because despite what the weights are connecting it to the next neurons, its activation is zero thus the input becomes zero. A bunch of zeros coming into the next neuron doesn't help it get into the positive domain and then these neurons' activations become zero too and the problem continues to cascade. Then we perform backprop and the gradients are zero so we don't update the weights and thus training halts. Not good.

We've talked about using leaky or parametric ReLUs or even the slower ELUs, but you can also lower your learning rates to help stop ReLU layers from not activating and thus dying. A large gradient, possibly due to too high of a learning rate, can update the weights in such a way that no datapoint will ever activate it again and since the gradient is zero we won't update the weight to something more reasonable so the problem will persist indefinitely.

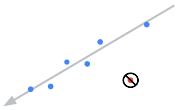
Proprietary + Confidential

There are benefits if feature values are small numbers



Roughly zero-centered, [-1, 1] range often works well

Small magnitudes help gradient descent converge and avoid NaN trap



Avoiding outlier values helps with generalization

Google Cloud

Let's have a quick intuition check. What will happen to a model if we have two useful signals, both independently correlated with the label, but that are on very different scales? For example, we might have a soup-deliciousness predictor, where features represent quantities of given ingredients. If the feature for chicken stock is measured in *litres*, but beef stock is measured in *millilitres*, then SGD might have a hard time converging well, since the optimal learning rate for these two dimensions is likely different.

Having your data clean and in a computationally helpful range has many benefits during the training process of your machine learning models. Having feature values small, and specifically zero centered, helps speed up training and avoids numerical issues. This is why batch normalization was helpful with exploding gradients because it made sure to keep not just the initial input features but all of the intermediate features within a healthy range as not to cause problems with our layers.

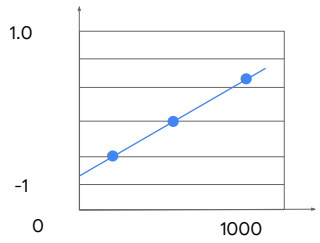
This also helps us avoid the NaN trap, where our model can blow up if values exceed numerical precision range. A combination of feature scaling and/or lower learning rate can help us avoid this nasty pitfall.

Also, avoiding outlier values helps with generalization, so detecting these, perhaps with anomaly detection, and preprocessing them out of our dataset before training can be a great help.

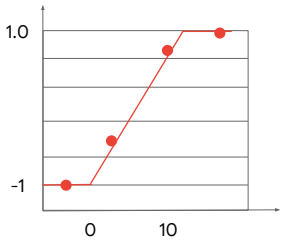
Remember, that there's no one best, one size fits all method for all data. It is possible

to think of good and bad cases for each of these approaches.

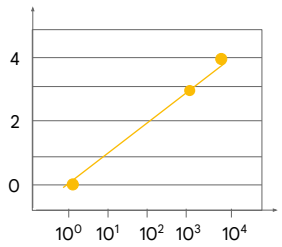
We can use standard methods to make feature values scale to small numbers



Linear scaling



Hard cap (clipping) to max, min



Log scaling

There are many methods to make our feature values scale to small numbers.

First there is linear scaling where you first find the minimum and maximum of the data. Then for each value, we subtract the minimum and then divide by the difference of the maximum and minimum. This will make all values between 0 and 1 where 0 will be the minimum and 1 will be the maximum. This is also called normalization.

There is also hard capping or clipping where you set a minimum value and a maximum value. For instance, if my minimum allowable value is -7 and my maximum allowable value is 10, then all values less than -7 will become -7 and all values greater than 10 will become 10.

Log scaling is another method where you apply the logarithm function to your input data. This is great when your data has a huge range and you want to condense it down to be more about just the magnitude of the value.
Another method which we talked about with batch normalization is standardization. Here you calculate the mean of your data and the standard deviation. Once you have these two values you subtract the mean from every datapoint and then divide by the standard deviation. This way your data becomes zero centered because your new mean becomes zero and your new standard deviation becomes one.
Of course there are many other ways to scale your data.

Quiz: Gradient Descent Debugging

Which of these is good advice if my model is experiencing exploding gradients?

A. Lower the learning rate

B. Add weight regularization

C. Add gradient clipping

D. Add batch normalization

E. See a doctor

F. C, D

G. A, C, D

H. A, B, C, D

Question: Which of these is good advice if my model is experiencing exploding gradients?

Quiz: Gradient Descent Debugging

Which of these is good advice if my model is experiencing exploding gradients?

A. Lower the learning rate

B. Add weight regularization

C. Add gradient clipping

D. Add batch normalization

E. See a doctor

F. C, D

G. A, C, D

H. A, B, C, D

Answer: The correct answer is A,B,C,D. The problem often occurs when weights get too large which can happen when our learning rate gets too high. This can lead to a whole bunch of other issues like numerical stability, divergence, and dying ReLUs. Therefore, lowering the learning rate to find that nice goldilocks zone is a great idea.

Weight regularization can also help in this respect because there will be a penalty for very large weights which should make it harder for gradients to explode.

Also, applying gradient clipping can ensure that gradients never get beyond a certain threshold that we set. This can help mitigate somewhat a high learning rate, however with a high enough rate it can still drive the weights to very high values.

Batch normalization can help keep the intermediate inputs at each layer stay within a tighter range so that there will be a much reduced chance of weights growing out of range for a small extra computational cost.

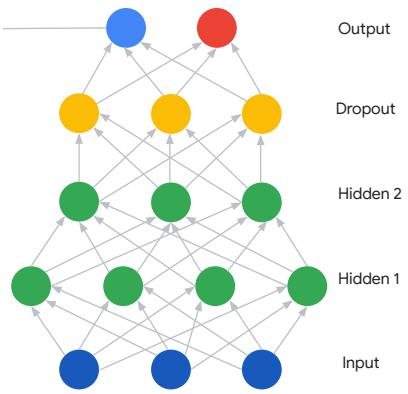
There are many methods to treat exploding gradients so you don’t need a doctor to help. All you have to do is experiment with these tools and see what works best.

Dropout layers are a form of regularization

During training only!
In prediction all nodes are kept.

Helps learn “multiple paths” -- think: ensemble models, random forests.

Dropout works by randomly “dropping out” unit activations in a network for a single gradient step.

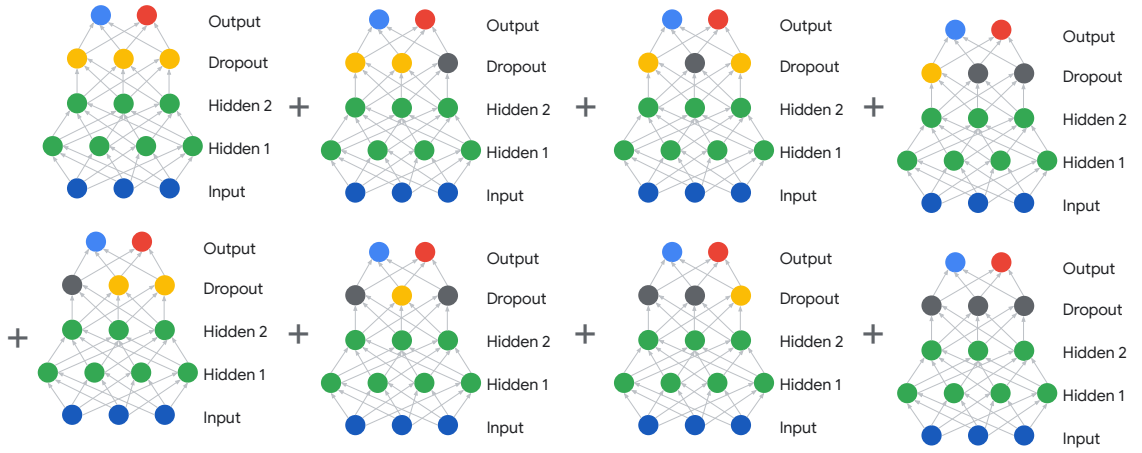


Another form of regularization that helps us build more generalizable models, is adding dropout layers to our neural networks. To use dropout, I add a wrapper to one or more of my layers. In TensorFlow the parameter you pass is called dropout which is the probability of dropping a neuron temporarily from the network, rather than keeping it turned on. You want to be careful when setting this number because for some other functions that have a dropout mechanism, they use keep probability which is the complement to drop probability. You wouldn’t want to intend only a 10% probability to drop, but actually are now only keeping 10% of your nodes randomly. That’s a very unintentional sparse model!

So how does dropout work under the hood? Let’s say we set a dropout probability of 20%. This means that for each forward pass of the network the algorithm will roll the dice for each neuron in the dropout wrapped layer. If the dice rolls greater than 20 then the neuron will stay active in the network, else the neuron will be “dropped” and output a value of zero regardless of its inputs, effectively not adding negatively or positively to the network since adding zero changes nothing and simulates that the neuron doesn’t exist. To make up for the fact that each node is only kept some percentage of the time, the activations are scaled by 1 / (1 - dropout probability) or in other words 1 / keep_prob during training so that it is the expectation value of the activation.

When not doing training, without having to change any code, the wrapper effectively disappears and the neurons in the formly dropout wrapped layer are always on and use whatever weights were trained by the model.

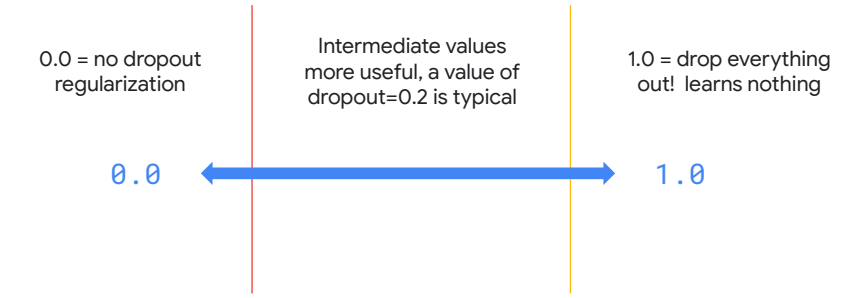
Dropout simulates ensemble learning



The awesome idea of dropout is it is essentially creating an ensemble model because for each forward pass there is effectively a different network that the mini-batch of data is seeing. When all of this is added together in expectation it is like I have trained 2^n neural networks, where n is the number of dropout neurons, and have them working in an ensemble similar to a bunch of decision trees working together in a random forest. There is also the added effect of spreading out the data distribution over the entire network rather than having the majority of the signal favor going along one branch of the network. I usually imagine this as diverting water in a river or stream with multiple shunts or dams to ensure all waterways eventually get some water and don't dry up. This way your network uses more of its capacity since the signal more evenly flows across the entire network and thus you will have better training and generalization without large neuron dependencies being developed in popular paths.

Typical values for dropout are between 20 to 50%. If you go much lower than that, then there is not much effect on the network since you are rarely dropping any nodes and if you go higher then training doesn't happen as well, since the network becomes too sparse to have the capacity to learn the data distribution. You also want to use this on larger networks because there is more capacity for the model to learn independent representations. In other words, there are more possible paths for the network to try.

The more you drop out, the stronger the regularization



The more you drop out, therefore the less you keep, the stronger the regularization. If you set your dropout probability to 1, then you keep nothing and every neuron in the wrapped dropout layer is effectively removed from the network and outputs a 0 activation. During backprop, this means that the weights will not update and this layer will learn nothing.

If you set your probability to 0, then all neurons are kept active and there is no dropout regularization. It's pretty much just a more computationally costly way to not have a dropout wrapper at all.

Of course somewhere between zero and one is where you want to be, specifically with dropout probabilities between 10 to 50% where a good baseline is usually starting at 20% and then adding more as needed. There is no one size fits all dropout probability for all models and data distributions.

Quiz: Dropout

Dropout acts as another form of _____. It forces data to flow down _____ paths so that there is a more even spread. It also simulates _____ learning. Don't forget to scale the dropout activations by the inverse of the _____. We remove dropout during _____.

A. Hyperparameter tuning, similar, deep, drop probability, training

B. Hyperparameter tuning, multiple, deep, drop probability, inference

C. Regularization, multiple, ensemble, keep probability, training

D. Regularization, multiple, ensemble, drop probability, inference

E. Regularization, multiple, ensemble, keep probability, inference

F. Hyperparameter tuning, multiple, deep, keep probability, inference

G. Regularization, similar, ensemble, keep probability, inference

Question: Dropout acts as another form of BLANK. It forces data to flow down BLANK paths so that there is a more even spread. It also simulates BLANK learning. Don't forget to scale the dropout activations by the inverse of the BLANK. We remove dropout during BLANK.

Quiz: Dropout

Dropout acts as another form of _____. It forces data to flow down _____ paths so that there is a more even spread. It also simulates _____ learning. Don't forget to scale the dropout activations by the inverse of the _____. We remove dropout during _____.

A. Hyperparameter tuning, similar, deep, drop probability, training

B. Hyperparameter tuning, multiple, deep, drop probability, inference

C. Regularization, multiple, ensemble, keep probability, training

D. Regularization, multiple, ensemble, drop probability, inference

E. Regularization, multiple, ensemble, keep probability, inference

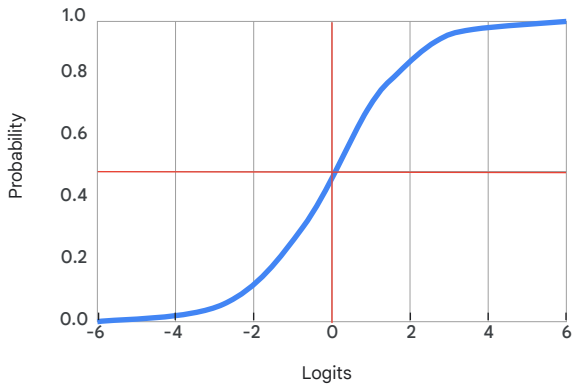
F. Hyperparameter tuning, multiple, deep, keep probability, inference

G. Regularization, similar, ensemble, keep probability, inference

Answer: The correct answer is E.

Dropout acts as another form of regularization so that the model can generalize better.
It does this turning off nodes with a dropout probability which forces data to flow down multiple paths so that there is a more even spread. Otherwise, data and the activations associated with it can learn to take preferential paths which might lead to undertraining of the network as a whole and provide poor performance on new data.
Dropout also simulates ensemble learning by create an aggregate of 2^n models due to the random turning off of nodes for each forward pass where n is the number of dropout nodes. Each batch sees a different network so that the model can't overfit on the entire training set much like a random forest.
Don't forget to scale the dropout activations by the inverse of the keep probability, which is 1 minus the dropout probability. We do this so that the expectation of the node will be scaled correctly during training since for inference they will always be on since we remove dropout during inference.

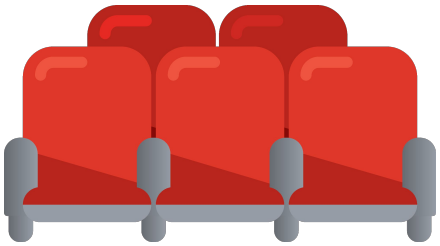
Logistic regression provides useful probabilities for binary-class problems



Here again is the sigmoid function which gives us calibrated probabilities. It is useful in logistic regression for binary class problems where I can find the probability of being in the positive class, where 1 minus that is the probability of being in the negative class. What then do we do when we have more than two classes?

There are lots of multi-class problems

How do we extend the logits idea to multi-class classifiers?



There are lots of multi-class problems. This example is of ticket types in an opera hall; perhaps the model is to predict the seat type to recommend. Let's say there are 4 places to sit: in the pit, in stalls, in the circle, or in a suite. If I want a probability for each of the seat types, I can't just use the normal binary classification because I have two too many classes. If pit is my positive class, then what is its negative class? What do I do with the remaining classes?

There are lots of multi-class problems

Model 1	
Positive Class	Negative Class
Pit	Other

Model 2	
Positive Class	Negative Class
Stalls	Other

Model 3	
Positive Class	Negative Class
Circle	Other

Model 4	
Positive Class	Negative Class
Suite	Other

One idea is to transform the problem from multi-class classification to many binary classification problems. A method to do this is the one vs. all or one vs. rest approach. In this approach, we will iterate through each class. For each iteration, that class will be the positive class and all the remaining classes will be lumped together into the negative class. In this way, I am predicting the probability of being in the positive class and conversely the probability of not being in the other classes. It is important that we output our predicted probability and not just the class label, so that we don't create ambiguities if multiple classes are predicted for a single sample. Once you have a model trained for each class being selected as the positive one, we move to the **most** valuable part of machine learning, predictions!

To make a prediction, you send your prediction example through each of the trained binary classification models. Then the model that produces the highest probability or confidence score you will choose as the overall predicted class.

Although this seems like a great solution, it comes with several problems. First, the scale of the confidence values might be different for each of the binary classification models which biases our overall prediction. However, even if that isn't the case, each of the binary classification models see very unbalanced data distributions since for each one, the negative class is the sum of all of the other classes besides the one that is currently marked the positive class.

There are lots of multi-class problems

Model 1	
Positive Class	Negative Class
Pit	Stalls

Model 2	
Positive Class	Negative Class
Pit	Circle

Model 3	
Positive Class	Negative Class
Pit	Suite

Model 4	
Positive Class	Negative Class
Stalls	Circle

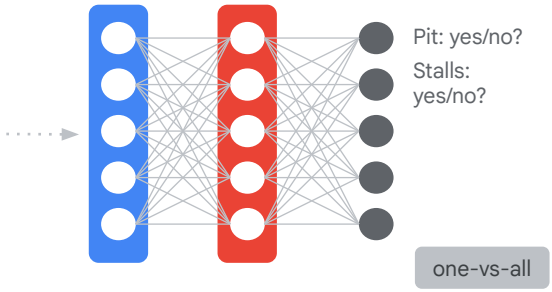
Model 5	
Positive Class	Negative Class
Stalls	Suite

Model 6	
Positive Class	Negative Class
Circle	Suite

A possible fix for this imbalance problem is the one vs one method. Here instead of having a model for each class, there is a model for each binary combination of the classes. If there are n classes, this means there would be n * (n-1) over 2 models so of order n squared. Already for 4 classes in our example that is 6 models, but if I had 1000 classes like with the Imagenet competition, there would be 499,500 models! Wow! Each model essentially outputs a vote for its predicted label, +1 or +0 for the positive class label of each model. Then all of the votes are accumulated and the class that has the most, wins! However, this still doesn't fix the ambiguity problem because based on the input distribution, it could end up having the same number of votes for different classes.

So, is there any way to do multi-class classification without these major drawbacks?

Idea:
Use separate output nodes for each possible class

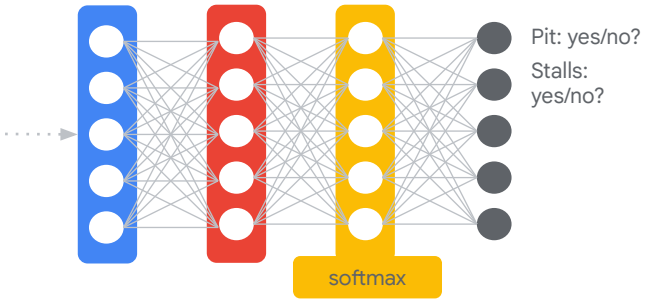


An idea could be to use the one vs all approach with neural networks, where instead of having multiple models for each class, there is one model with a unique output for each possible class. We can train this model on a signal of “my class” vs “all other classes” for each example that it sees. Therefore, we have to be careful in how we design our labels. Instead of having just a 1 for our true class, we will have a vector of length of the number of classes where our true class’ corresponding element will be 1 and the rest will be zero. This way we will reward the corresponding sigmoid neuron for the true class if it gets close to 1 and we will punish the other sigmoid neurons if they also get close to 1 with a higher error to be backpropagated back through the network to update their weights.

However, we may have problems with millions of unique classes, since we will have millions of output neurons, thus millions of loss calculations, followed by millions of errors being backpropagated through the network. Very computationally expensive! Is there a better way?

Add additional constraint, that total of outputs = 1.0

$$p(y = j \mid \mathbf{x}) = \frac{exp(\mathbf{w}_j^T \mathbf{x} + b_j)}{\sum_{k \in K_j} exp(\mathbf{w}_k^T \mathbf{x} + b_k}$$



If we simply add an additional constraint, that the sum of outputs equals one, then it allows the outputs to be interpreted as probabilities. This normalizing function is called softmax. At each node we find the exponential of w x plus b and then divide by the sum of all of the nodes. This ensures all nodes are between 0 and 1 and that the total probability equals 1 as it should. This way for each example, you will get a normalized probability for each class, where you can then take the argmax to find the class with the highest probability as your predicted label.

Use one softmax loss for all possible classes

```
logits = tf.matmul(...) # logits for each output node
                           -> shape = [batch_size, num_classes]
labels = …              # one-hot encoding in [0, num_classes)
                           -> shape = [batch_size, num_classes]
loss = tf.reduce_mean(
   tf.nn.softmax_cross_entropy_with_logits_v2(
                                     logits, labels) -> shape = [batch_size]
)
```


In TensorFlow, we calculate our logits in our final layer as the matrix multiplication of w and x with a bias node added to the result if one exists. This will give us a tensor shape of batch size by the number of classes. Our labels are one hot encoded as we talked about previously where the true class gets a 1 and the other classes get 0 for each example, therefore also having shape a tensor of batch size by the number of classes. Note, because we are using TensorFlow's softmax cross entropy with logits function, the labels can actually be soft. What I mean by this is, even though the classes are still mutually exclusive, their probabilities need not be. If you had three classes, for example, your mini-batch could have one of its labels be 0.15, 0.8, and 0.05 as your label. They are not one hot encoded, however they still are a valid probability distribution since they sum to 1.

Finally, we compare our logits with our labels using softmax cross entropy with logits and we will get a resultant tensor of shape batch size. In TensorFlow 1.5+, a version 2 of the function was created with the version 1 function set to be deprecated. To get the average loss for that mini-batch, just use reduce mean on the output.

Use one softmax loss for all possible classes

```
logits = tf.matmul(...) # logits for each output node
                          -> shape = [batch_size, num_classes]
labels = …              # index in [0, num_classes)
                          -> shape = [batch_size]
loss = tf.reduce_mean(
   tf.nn.sparse_softmax_cross_entropy_with_logits(
                                      logits, labels) -> shape = [batch_size]
)
```


For convenience, TensorFlow has another function you can use instead to calculate the softmax called sparse softmax cross entropy with logits. In this case we do away with the one hot or soft encoding of our labels and instead just provide the index of the true class between 0 and the number of classes minus 1. This means that our labels are now a tensor of shape batch size. The output of the function is still the same as before as a tensor of shape batchsize and I will still just reduce mean that tensor to get the average loss of the mini-batch.

Use softmax only when classes are mutually exclusive

```
tf.nn.sigmoid_cross_entropy_with_logits(
                          logits, labels) -> shape = [batch_size, num_classes]
```

"Multi-Class, Single-Label Classification"
An example may be a member of only one class.

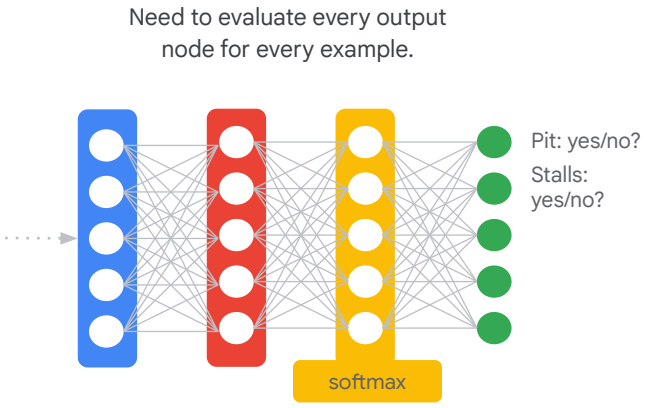
Are there multi-class settings where examples may belong to more than one class?

Remember, for both softmax functions, we are only using them because our classes are mutually exclusive. For instance, image 1 is only a picture of a dog and image 2 is only a picture of a cat.

However, what if image 3 is a picture of both a dog and a cat and for my ML problem I want to know that? Using softmax, I will get a probability for each one, but I will take the argmax of it as my label. Therefore depending on my image and model it might label it as dog or it might label it as cat. **This is no good because** I want to know if both are in there and if there are any of my other classes in there as well.

This is a multi-class, multi-label classification problem. In this case I want a probability of each class from 0 to 1. Fortunately, TensorFlow has a nifty function that does just that called sigmoid cross entropy with logits which returns a batch size by number of classes tensor.

If you have hundreds or thousands of classes, loss computation can become a significant bottleneck



We need to evaluate every output node for every example. Of course “every output node” means also the weights that lead up to it. So, a single step of a 100-output node network is like 100 steps of a single-output network. Hugely expensive and hard to scale for a very large number of classes. We need some way to approximate this softmax so that we can reduce some of the computational cost for very large multi-class problems.

Approximate versions of softmax exist

Candidate Sampling

calculates for all the positive labels, but only for a random sample of negatives:

```
tf.nn.sampled_softmax_loss
```

Noise-contrastive

approximates the denominator of softmax by modeling the distribution of outputs:

```
tf.nn.nce_loss
```


Fortunately, approximate versions of softmax exist.

Candidate sampling calculates for all of the positive labels but rather than also performing the computation on ALL of the negative labels, it randomly samples some negatives which should greatly reduce computation. The number of negatives sampled is an important hyperparameter to a CandidateSampling model. It is always, for obvious reasons, an underestimate. In TensorFlow, we can use the function sampled softmax loss.

Another way to approximate the softmax is to use noise constrastive estimation. Noise constrastive estimation approximates the denominator of the softmax, which contains the sum of all exponentials of the logits, by modeling the distribution of outputs instead. This can provide an approximate, less computationally expensive means to find our softmax loss without having to evaluate every class in the sum in the denominator.

Candidate sampling is more intuitive and doesn't require a really good model. Noise-contrastive requires a really good model since it relies on modeling the distribution of the outputs.

Typically we will use these functions during training, but for evaluation and inference, for better accuracy we usually use the full softmax. To do this, make sure to change the default partition strategy from mod to div for the losses to be consistent between training, evaluation, and prediction.

Quiz: Softmax

For our classification output, if we have both mutually exclusive labels and probabilities, we should use _____. If the labels are mutually exclusive, but the probabilities aren't, we should use _____. If our labels aren't mutually exclusive, we should use _____.

I. tf.nn.sigmoid_cross_entropy_with_logits
II. tf.nn.sparse_softmax_cross_entropy_with_logits
III. tf.nn.softmax_cross_entropy_with_logits_v2

A. III, II, I

B. I, II, III

C. III, I, II

D. II, III, I

Question: For our classification output, if we have both mutually exclusive labels and probabilities, we should use BLANK. If the labels are mutually exclusive, but the probabilities aren't then we should use BLANK. If our labels aren't mutually exclusive, we should use BLANK.

Quiz: Softmax

For our classification output, if we have both mutually exclusive labels and probabilities, we should use _____. If the labels are mutually exclusive, but the probabilities aren't, we should use _____. If our labels aren't mutually exclusive, we should use _____.

I. tf.nn.sigmoid_cross_entropy_with_logits
II. tf.nn.sparse_softmax_cross_entropy_with_logits
III. tf.nn.softmax_cross_entropy_with_logits_v2

A. III, II, I

B. I, II, III

C. III, I, II

D. II, III, I

Answer: The correct answer is D.

For our classification output, if we have both mutually exclusive labels and probabilities, we should use sparse_softmax_cross_entropy_with_logits. This doesn't allow for soft labels, but does help reduce the model data size since you can compress your labels into just being the index of the true class rather than a vector of the number of classes for each example.

If the labels are mutually exclusive, but the probabilities aren't then we should use softmax_cross_entropy_with_logits_v2. This means that there is only one true class for each example and we allow for soft labels where the true class does not need to one-hotted for the true class but can be any combination of values between 0 and 1 for each class as long as they all sum up to 1.

If our labels aren't mutually exclusive, we should use sigmoid.cross_entropy_with_logits. This way we will get a probability for each possible class which can give us confidence scores of each class being represented in the output such as an image with multiple classes in it where we want to know the existence of each class.