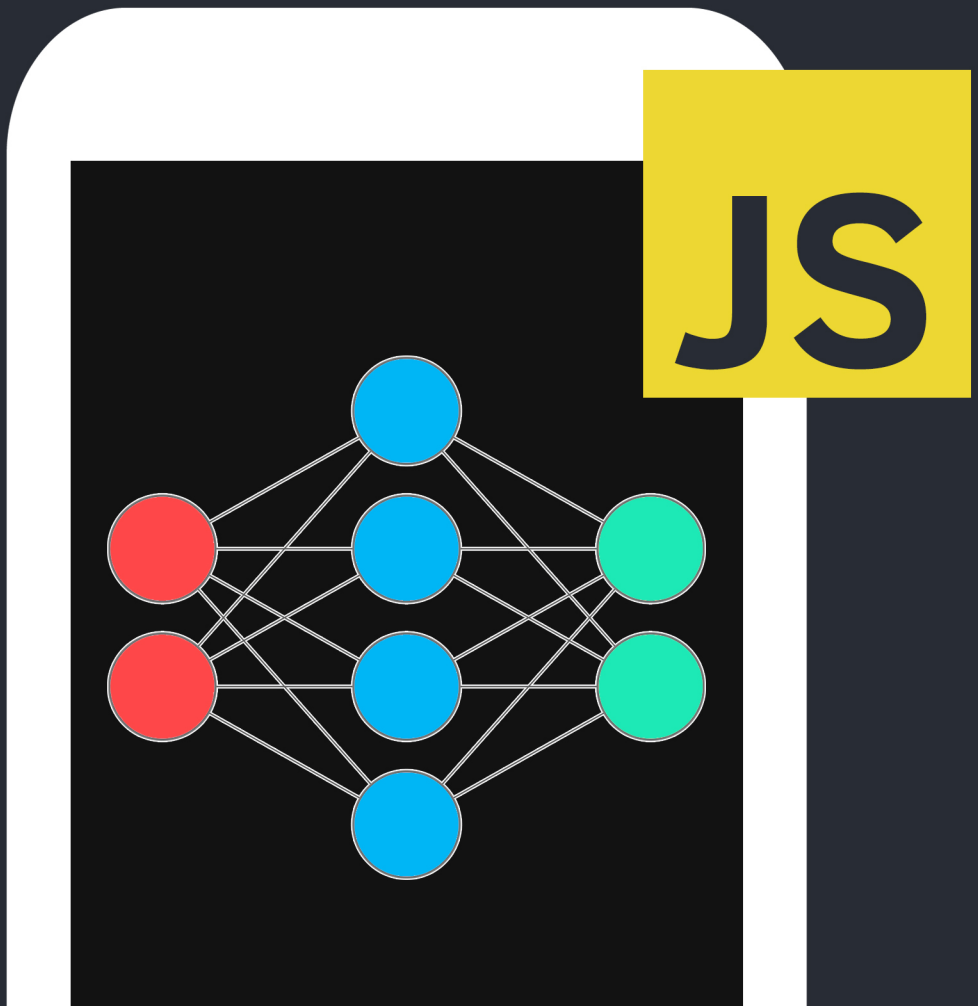
Venelin Valkov's

Deep Learning for JavaScript Hackers



Deep Learning for JavaScript Hackers

Beginners guide to understanding Machine Learning in the browser with TensorFlow.js

Venelin Valkov

This book is for sale at http://leanpub.com/deep-learning-for-javascript-hackers

This version was published on 2019-10-07

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By Venelin Valkov

Hands-On Machine Learning from Scratch

Contents

Getting Started with TensorFlow.js

> TL;DR Learn about the basics of Machine Learning with TensorFlow.js - Tensors, basic visualizations and train a simple model that converts kilograms to pounds

So what is this thing TensorFlow.js?

> TensorFlow.js[1] is a library for developing and training ML models in JavaScript, and deploying in browser or on Node.js

For our purposes, TensorFlow.js will allow you to build Machine Learning models (especially Deep Neural Networks) that you can easily integrate with existing or new web apps. Think of your ReactJs, Vue, or Angular app enhanced with the power of Machine Learning models.

Run the complete source code for this tutorial right in your browser[2]

Tensors

Tensors are the main building blocks of TensorFlow. They are n-dimensional data containers. You can think of them as multidimensional arrays in languages like PHP, JavaScript, and others. What that means is that you can use tensors as a scalar, vector, and matrix values, since they are a generalization of those.

Each Tensor contains the following properties

- `rank` - number of dimensions
- `shape` - size of each dimension
- `dtype` - data type of the values

Let's start by creating your first Tensor:

```
1  import * as tf from "@tensorflow/tfjs";
2
3  const t = tf.tensor([1, 2, 3]);
```

Check it's rank:

[1] https://www.tensorflow.org/js
[2] https://codesandbox.io/s/getting-started-tensorflow-js-3182j?fontsize=14

```
console.log(t.rank);
```

```
1
```

That confirms that your Tensor is 1-dimensional. Let's check the shape:

```
console.log(t.shape);
```

```
[3]
```

1-dimensional with 3 values. But how can you see the contents of this thing?

```
console.log(t);
```

```
Tensor {kept: false, isDisposedInternal: false â€¦}
```

Not what you've expected, right? Tensors are custom objects and have a print()[3] method that output their values:

```
t.print();
```

```
Tensor
    [1, 2, 3]
```

Of course, the values don't have to be just numeric. You can create tensors of strings:

```
const st = tf.tensor(["hello", "world"]);
```

You can use tensor2d()[4] to create matrices (or 2-dimensional tensors):

```
const t2d = tf.tensor2d([[1, 2, 3], [4, 5, 6]]);
console.log(t2d.shape);
```

[3] https://js.tensorflow.org/api/latest/#tf.Tensor.print
[4] https://js.tensorflow.org/api/latest/#tensor2d

```
[2, 3]
```

There are some utility methods that will be handy when we start developing models. Let's start with ones()[5]:

```
tf.ones([3, 3]).print();
```

```
Tensor
    [[1, 1, 1],
     [1, 1, 1],
     [1, 1, 1]]
```

You can use reshape()[6] to change the dimensionality of a Tensor:

```
tf.tensor([1, 2, 3, 4, 5, 6])
  .reshape([2, 3])
  .print();
```

```
Tensor
    [[1, 2, 3],
     [4, 5, 6]]
```

Tensor Math

You can use add()[7] to do element-wise addition:

```
const a = tf.tensor([1, 2, 3]);
const b = tf.tensor([4, 5, 6]);

a.add(b).print();
```

```
Tensor
    [5, 7, 9]
```

and dot()[8] to compute the dot product of two tensors:

[5] https://js.tensorflow.org/api/latest/#ones
[6] https://js.tensorflow.org/api/latest/#reshape
[7] https://js.tensorflow.org/api/latest/#add
[8] https://js.tensorflow.org/api/latest/#dot

```
const d1 = tf.tensor([[1, 2], [1, 2]]);
const d2 = tf.tensor([[3, 4], [3, 4]]);
d1.dot(d2).print();
```

```
Tensor
    [[9, 12],
     [9, 12]]
```

Finally, let's have a look at transpose()[9]:

```
tf.tensor([[1, 2], [3, 4]])
  .transpose()
  .print();
```

```
Tensor
    [[1, 3],
     [2, 4]]
```

You can think of the transpose as a flipped-axis version of the input Tensor.

Have a look at all arithmetic operations[10]

Visualization with tfjs-vis

> tfjs-vis[11] is a small library for in browser visualization intended for use with TensorFlow.js.

Let's start by creating a simple bar chart. Here's what our data looks like:

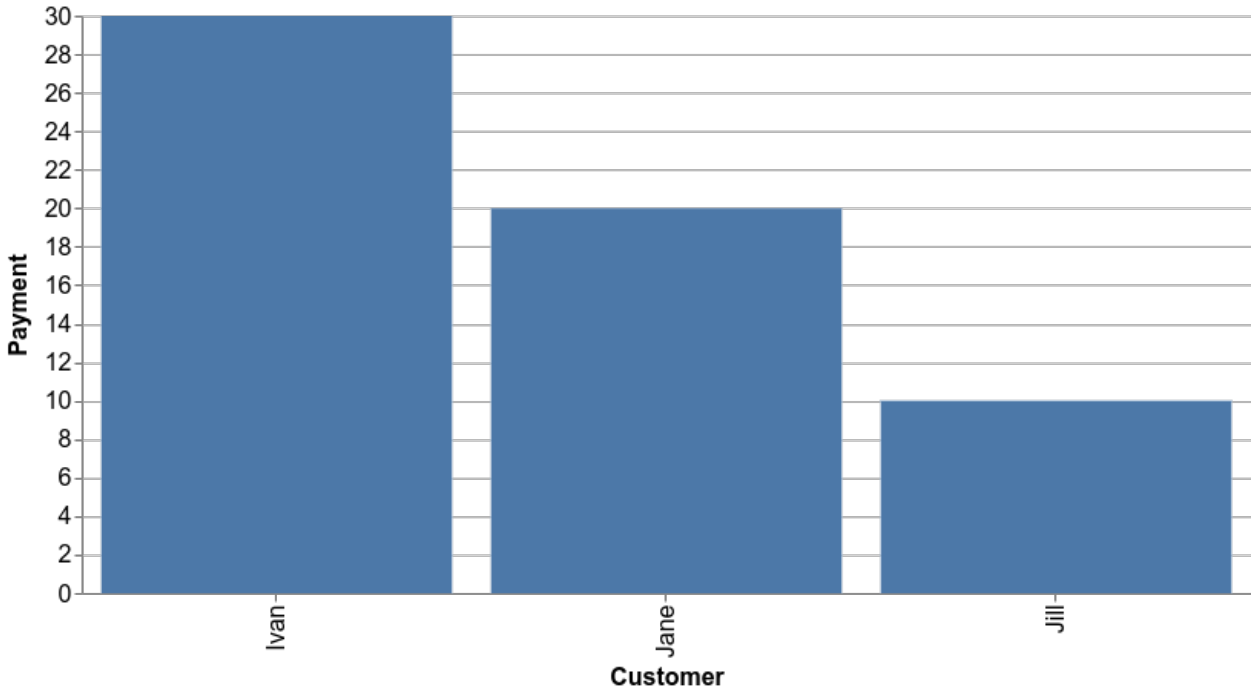
```
import * as tfvis from "@tensorflow/tfjs-vis";

const data = [
  { index: "Jill", value: 10 },
  { index: "Jane", value: 20 },
  { index: "Ivan", value: 30 }
];
```

Now, let's render the data using barchart()[12]:

[9] https://js.tensorflow.org/api/latest/#transpose
[10] https://js.tensorflow.org/api/latest/#Operations-Arithmetic
[11] https://github.com/tensorflow/tfjs-vis
[12] https://js.tensorflow.org/api_vis/latest/#render.barchart

```
const container = document.getElementById("barchart-cont");
tfvis.render.barchart(container, data, {
  xLabel: "Customer",
  yLabel: "Payment",
  height: 350,
  fontSize: 16
});
```

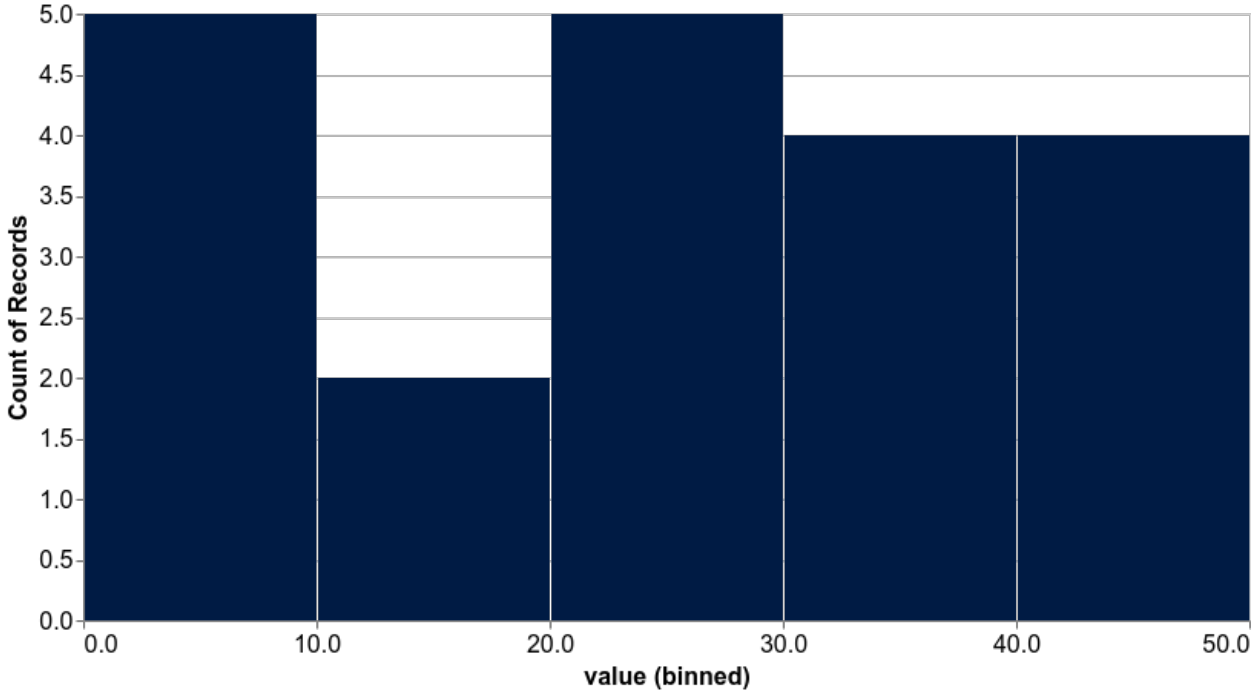
Note that we provide a DOM element to the renderer as a container for our chart, which might be handy when you want to embed the charts in your apps.



Let's have a look at histogram()[13] and create a sample chart:

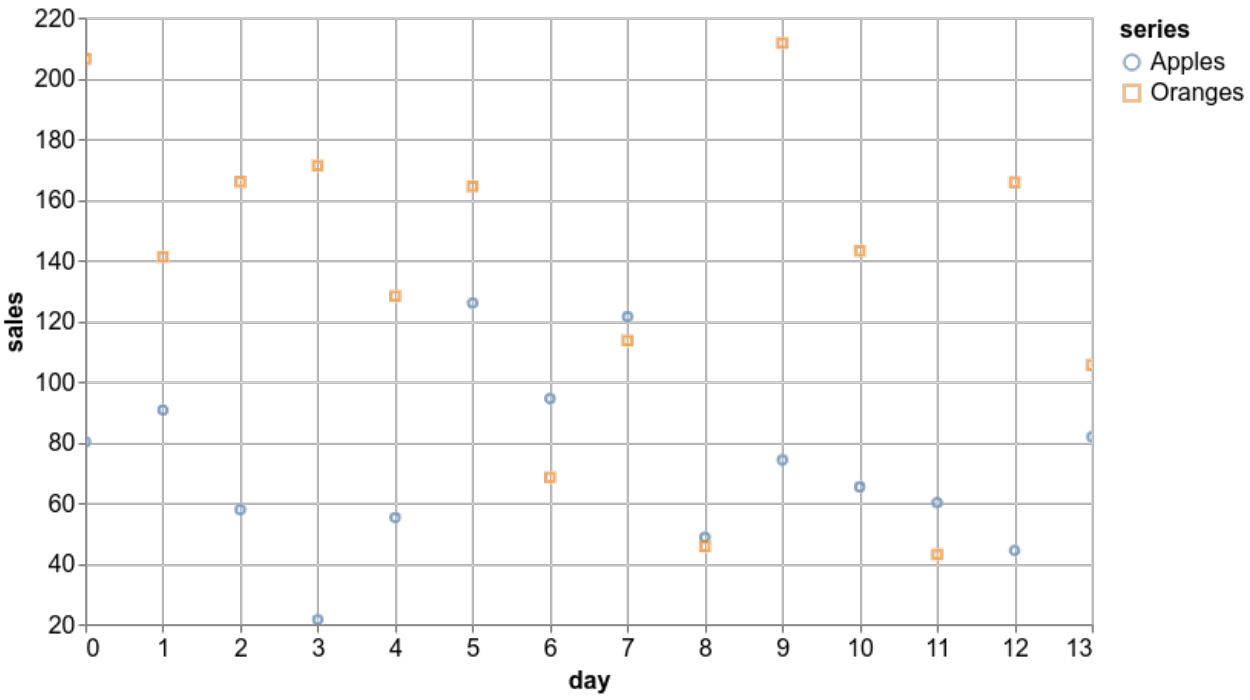
```
const data = Array(20)
  .fill(0)
  .map(x => Math.random() * 50);

const container = document.getElementById("histogram-cont");
tfvis.render.histogram(container, data, {
  maxBins: 5,
  height: 450,
  fontSize: 16
});
```

[13] https://js.tensorflow.org/api_vis/latest/#render.histogram



The API is pretty consistent for those 2 charts. Let's do a scatter plot:

```
const apples = Array(14)
  .fill(0)
  .map(y => Math.random() * 100 + Math.random() * 50)
  .map((y, x) => ({ x: x, y: y }));

const oranges = Array(14)
  .fill(0)
  .map(y => Math.random() * 100 + Math.random() * 150)
  .map((y, x) => ({ x: x, y: y }));

const series = ["Apples", "Oranges"];

const data = { values: [apples, oranges], series };

const container = document.getElementById("scatter-cont");
tfvis.render.scatterplot(container, data, {
  xLabel: "day",
  yLabel: "sales",
  height: 450,
  zoomToFit: true,
  fontSize: 16
});
```



Have a look at the complete tfjs-vis API[14]

Train your first model

Time to put what you've learned into practice and build your first model. To make it somewhat realistic, we'll try to approximate the conversion of kgs to lbs, which is described by this function:

```
1  const kgToLbs = kg => kg * 2.2;
```

Let's use it to prepare our data and create 2000 training examples:

```
1  const xs = tf.tensor(Array.from({ length: 2000 }, (x, i) => i));
2  const ys = tf.tensor(Array.from({ length: 2000 }, (x, i) => kgToLbs(i)));
```

We're going to use a style of Machine Learning known as Supervised Learning[15]. In a nutshell, we need to provide 2 arrays to our model - X is the training features (kilograms), and y is the training labels (corresponding pounds).

TensorFlow.js allows you to build layered models using sequential()[16]. We're going to go extremely simple: 1 layer, input size of 1, and 1 learning parameter:

[14] https://js.tensorflow.org/api_vis/latest/
[15] https://en.wikipedia.org/wiki/Supervised_learning
[16] https://js.tensorflow.org/api/latest/#sequential

```
const model = tf.sequential();

model.add(tf.layers.dense({ units: 1, inputShape: 1 }));
```

and teach it to convert kilograms to pounds:

```
model.compile({
  loss: "meanSquaredError",
  optimizer: "adam"
});

await model.fit(xs, ys, {
  epochs: 100,
  shuffle: true
});
```

Your model needs a metric to know how well is doing. In our case that is Mean Squared Error (MSE)[17]. Once you know how to measure the error, you need something to know how to minimize it using the data. In our case, that is the Adam optimizer[18].

Finally, we use the data to train our model for *100* epochs (number of times our model sees the data) and request to shuffle it. Why shuffle? We don't want our model to learn the ordering of the data, just the relationship between different examples.

After the training is complete (might take some time) you can use your model to predict what amount of pounds correspond to 10 kg:

```
const lbs = model
  .predict(tf.tensor([10]))
  .asScalar()
  .dataSync();

console.log("10 kg to lbs: " + lbs);
```

```
10 kg to lbs: 22.481597900390625
```

Seems to be doing good, right?

[17] https://en.wikipedia.org/wiki/Mean_squared_error
[18] https://js.tensorflow.org/api/latest/#train.adam

Conclusion

Congratulation on finishing the first part of your journey to Machine Learning understanding. You learned about:

- Tensors: n-dimensional data containers
- tfjs-vis: visualization library integrated with TensorFlow.js
- predict pounds from kilograms using a simple model

Run the complete source code for this tutorial right in your browser[19]

I hope that this tutorial just made you thirsty for knowledge about what is possible with Machine Learning and JavaScript. Ready for the next one?

[19] https://codesandbox.io/s/getting-started-tensorflow-js-3182j?fontsize=14

Predicting Diabetes using Logistic Regression

> TL;DR Build a Logistic Regression model in TensorFlow.js using the high-level layers API, and predict whether or not a patient has Diabetes. Learn how to visualize the data, create a Dataset, train and evaluate multiple models.

You've been living in this forgotten city for the past 8+ months. You never felt comfortable anywhere but home. However, this place sets a new standard. The constant changes between dry and humid heat are killing you, fast.

The Internet connection is spotty at best, and you haven't heard from your closed ones for more than two weeks. You have no idea how your partner is and how your kids are doing. You sometimes question the love for your country.

This morning you feel even worse. Constantly hungry and thirsty. You urinated four times, already, and your vision is somewhat blurry. It is not just today you were feeling like that for a week, at least.

You went to the doctor, and she said you might be having Diabetes. Both your mother and father suffer from it, so it seems likely to you. She wasn't that sure and did a glucose test. Unfortunately, you're being called and should go before the results are in.

You're going away for two weeks. Only a couple of guys and your laptop! You have a couple of minutes and download a Diabetes patient dataset. You have TensorFlow.js already installed and a copy of the whole API. Can you build a model to predict whether or not you have Diabetes?

Run the complete source code for this tutorial right in your browser[20]

Diabetes data

> **Diabetes mellitus (DM)**[21], commonly known as diabetes, is a group of metabolic disorders characterized by high blood sugar levels over a prolonged period. Symptoms of high blood sugar include frequent urination, increased thirst, and increased hunger. If left untreated, diabetes can cause many complications. Acute complications can include diabetic ketoacidosis, hyperosmolar hyperglycemic state, or death. Serious long-term complications include cardiovascular disease, stroke, chronic kidney disease, foot ulcers, and damage to the eyes.

[20] https://codesandbox.io/s/logistic-regression-tensorflow-js-r6b5m?fontsize=14
[21] https://en.wikipedia.org/wiki/Diabetes

> As of 2017, an estimated 425 million people had diabetes worldwide (around 5.5%)

Our data comes from Kaggle[22] but was first introduced in the paper: Using the ADAP Learning Algorithm to Forecast the Onset of Diabetes Mellitus[23]

> The population for this study was the Pima Indian population near Phoenix, Arizona. That population has been under continuous study since 1965 by the National Institute of Diabetes and Digestive and Kidney Diseases because of its high incidence rate of diabetes. Each community resident over 5 years of age was asked to undergo a standardized examination every two years, which included an oral glucose tolerance test. Diabetes was diagnosed according to World Health Organization Criteria; that is, if the 2 hour post-load plasma glucose was at least 200 mg/dl (11.1 mmol/l) at any survey examination or if the Indian Health Service Hospital serving the community found a glucose concentration of at least 200 mg/dl during the course of routine medical care.

Here is a summary of the data:

- `Pregnancies` - Number of times pregnant
- `Glucose` - Plasma glucose concentration a 2 hours in an oral glucose tolerance test
- `BloodPressure` - Diastolic blood pressure (mm Hg)
- `SkinThickness` - Triceps skin fold thickness (mm)
- `Insulin` - 2-Hour serum insulin (mu U/ml)
- `BMI` - Body mass index ($\frac{weight}{height^2}$ in kg/m)
- `DiabetesPedigreeFunction` - Diabetes Pedigree Function (DPF)
- `Age` - Age (years)
- `Outcome` - Class variable (0 - healthy or 1 - diabetic)

According to Estimating Probabilities of Diabetes Mellitus Using Neural Networks[24] paper, the DPF provides:

> A synthesis of the diabetes mellitus history in relatives and the genetic relationship of those relatives to the subject. The DPF uses information from parents, grandparents, siblings, aunts and uncles, and first cousins. It provides a measure of the expected genetic influence of affected and unaffected relatives on the subject's eventual diabetes risk.

[22] https://www.kaggle.com/uciml/pima-indians-diabetes-database
[23] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2245318/pdf/procascamc00018-0276.pdf
[24] http://www.personal.kent.edu/~mshanker/personal/Zip_files/sar_2000.pdf

Who are Pima Indians?

> The Pima (or Akimel O'odham, also spelled Akimel O'otham, "River People", formerly known as Pima) are a group of Native Americans living in an area consisting of what is now central and southern Arizona. The majority population of the surviving two bands of the Akimel O'odham are based in two reservations: the Keli Akimel O'otham on the Gila River Indian Community (GRIC) and the On'k Akimel O'odham on the Salt River Pima-Maricopa Indian Community (SRPMIC).

Read the data

We'll use the Papa Parse[25] library to read the csv file. Unfortunately, Papa Parse doesn't work well with `await/async`. Let's change that:

[25] https://www.papaparse.com/

```
import * as Papa from "papaparse";

Papa.parsePromise = function(file) {
  return new Promise(function(complete, error) {
    Papa.parse(file, {
      header: true,
      download: true,
      dynamicTyping: true,
      complete,
      error
    });
  });
};
```

We use the `dynamicTyping` parameter to instruct Papa Parse to convert the numbers in the dataset from strings. Let's define a function that loads the data:

```
const loadData = async () => {
  const csv = await Papa.parsePromise(
    "https://raw.githubusercontent.com/curiousily/Logistic-Regression-with-TensorFlo\
w-js/master/src/data/diabetes.csv"
  );
  return csv.data;
};
```

and use it:

```
const data = await loadData();
```

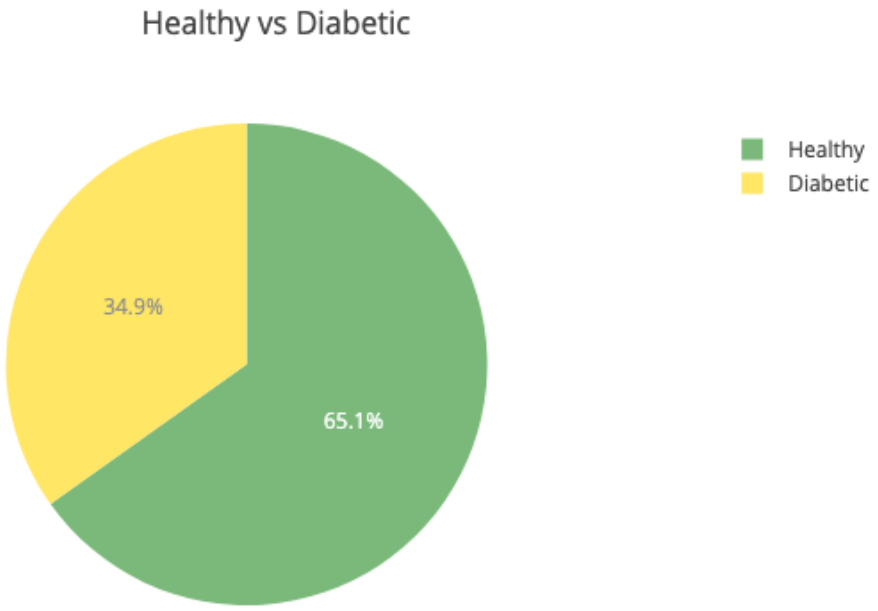
Good job! We have the data, let get familiar with it!

Exploration

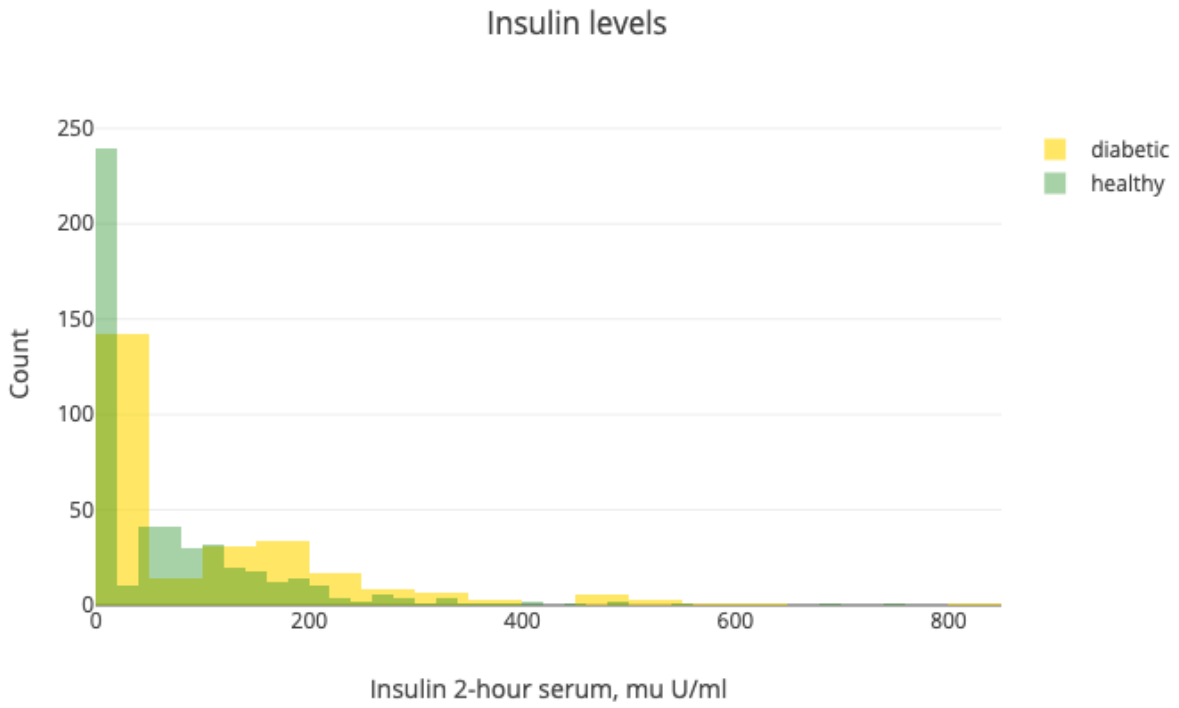
While tfjs-vis[26] is nice and well integrated with TensorFlow.js, it lacks (at the time of this writing) a ton of features you might need - overlay plots, color changes, scale customization, etc. That's why we'll use Plotly's Javascript library[27] to make some beautiful plots for our data exploration.

Let's have a look at the distribution of healthy vs diabetic people:

[26] https://github.com/tensorflow/tfjs-vis
[27] https://github.com/plotly/plotly.js/

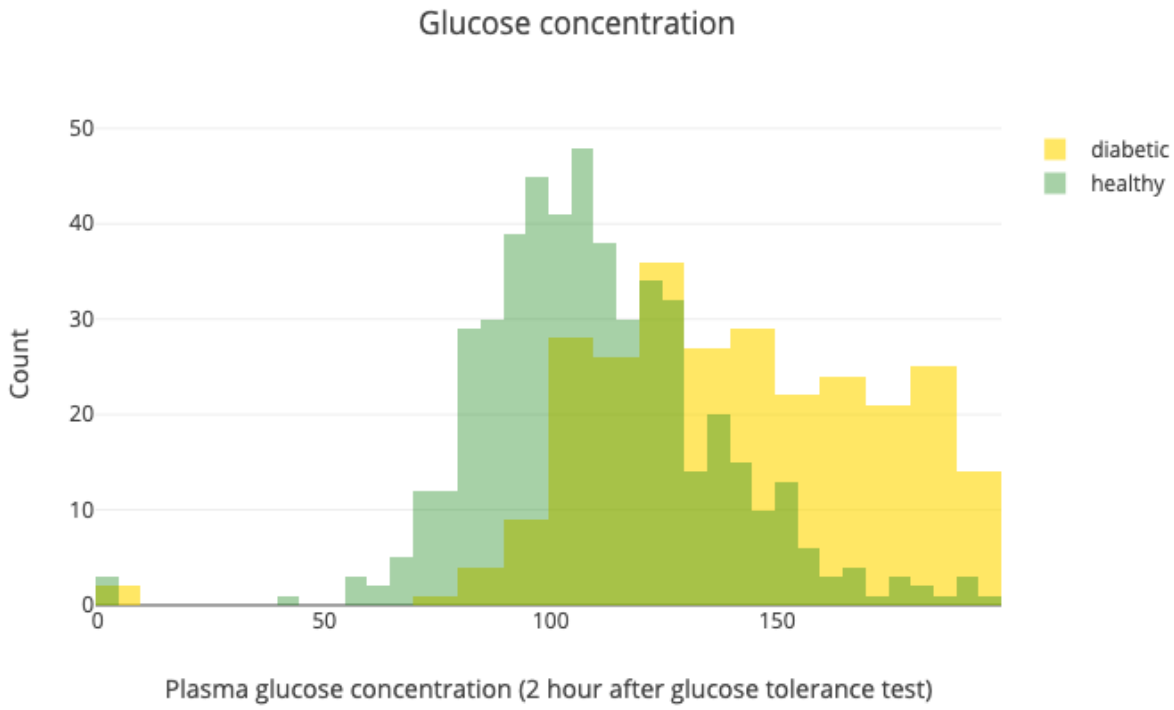


Little above 65% of the patients in our dataset are healthy. That means that our model should be more accurate than 65% of the time, to be any good. Next up - the insulin levels:



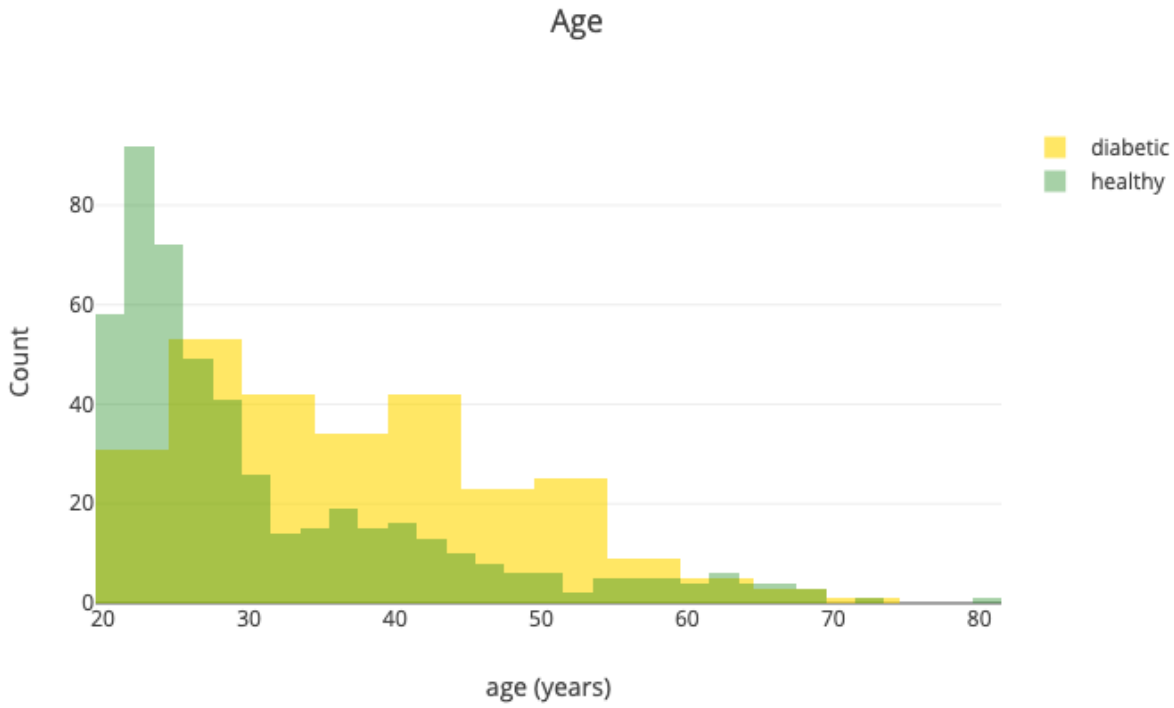
Note that there is a big overlap between the two distributions. Also, we have a lot of 0s in the dataset. Seems like we have a lot of missing values. NaNs are replaced with 0s.

Another important one is the glucose levels after the test:



While there is some overlap, this test seems like it separates the healthy from diabetic patients pretty well.

Let's have a look at the age:

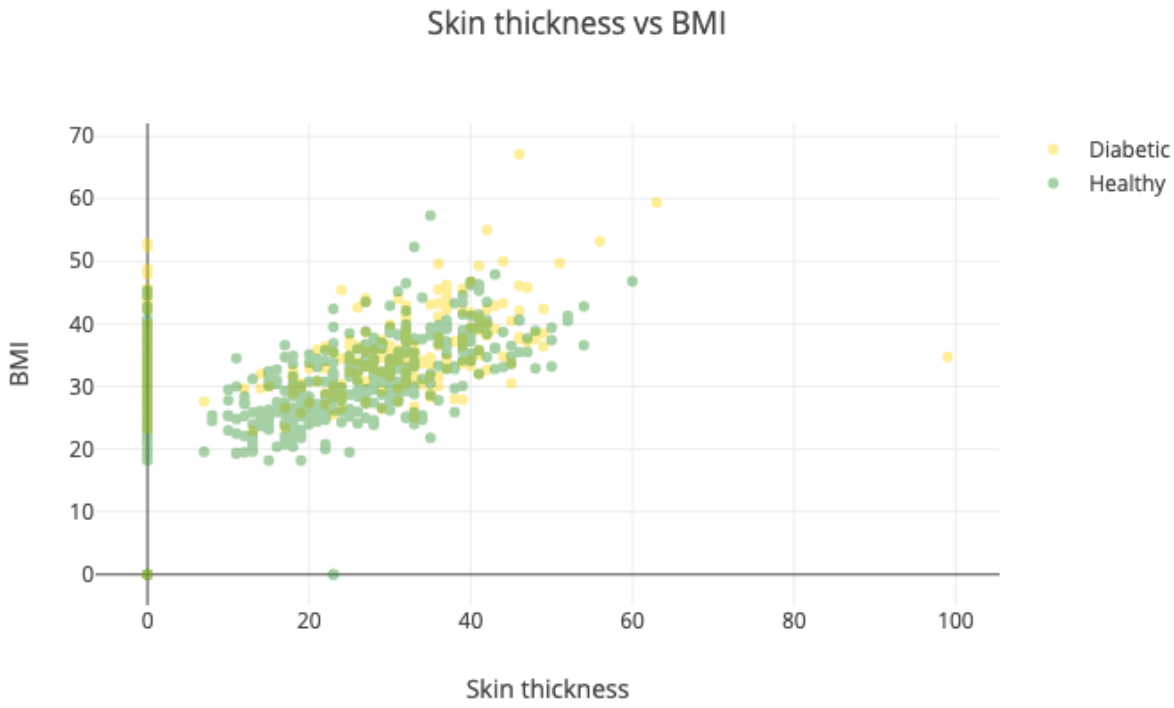


Generally speaking, it seems like older people are more likely to have diabetes.

Maybe we should take a look at the relationship between age and glucose levels:

The combination of those two seems to separate healthy and diabetic patients very well. That might do wonders for our model.

Another combination you might want to try is the skin thickness vs BMI:



Yep, this one is horrible and doesn't tell us much :)

Preprocessing

Currently, our data sits in an array of objects. Unfortunately, TensorFlow doesn't work well with those. Luckily, there is the tfjs-data[28] package. We're going to create a Dataset from our CSV file and use it to train our model with the `createDatasets()` function:

```
const createDataSets = (data, features, testSize, batchSize) => {
  ...
};
```

The `features` parameter specifies which columns are in the dataset. `testSize` is the fraction of the data that is going to be used for testing. `batchSize` controls the number of data points when the dataset is split into chunks (batches).

Let's start by extracting the features from the data:

[28] https://github.com/tensorflow/tfjs-data

```
const X = data.map(r =>
  features.map(f => {
    const val = r[f];
    return val === undefined ? 0 : val;
  })
);
```

We're replacing missing values in our features with 0s. You might try to train your model without this step and see what happens?

Let's prepare the labels:

```
const y = data.map(r => {
  const outcome = r.Outcome === undefined ? 0 : r.Outcome;
  return oneHot(outcome);
});
```

Here's the definition of oneHot:

```
const oneHot = outcome => Array.from(tf.oneHot(outcome, 2).dataSync());
```

One-hot encoding turns categorical variables (healthy - 0 and diabetic - 1) into an array where 1 corresponds to the position of the category and all other variables are 0. Here are some examples:

```
1; // diabetic =>
[0, 1];
```

```
0; // healthy =>
[1, 0];
```

Let's create a Dataset[29] from our data:

```
const ds = tf.data
  .zip({ xs: tf.data.array(X), ys: tf.data.array(y) })
  .shuffle(data.length, 42);
```

Note that we also shuffle the data with a seed of **42** :)

Finally, let's split the data into training and validation datasets:

[29] https://js.tensorflow.org/api/latest/#class:data.Dataset

```
const splitIdx = parseInt((1 - testSize) * data.length, 10);

return [
  ds.take(splitIdx).batch(batchSize),
  ds.skip(splitIdx + 1).batch(batchSize),
  tf.tensor(X.slice(splitIdx)),
  tf.tensor(y.slice(splitIdx))
];
```

We use take[30] to create the training dataset, skip[31] to omit the training examples for the validation dataset and finally, split the data into chunks using batch[32].

Additionally, we return data for testing our model (more on this later).

Logistic Regression

Logistic Regression (contrary to its name) allows you to get binary (yes/no) answers from your data. Moreover, it gives you the probability for each answer. Questions like:

- Is this email spam?
- Should I ask my boss for a higher salary?
- Does this patient have diabetes?
- Is this person a real friend?
- Does my partner cheat on me?
- Do I cheat on my partner?
- Do you get where I am getting at?

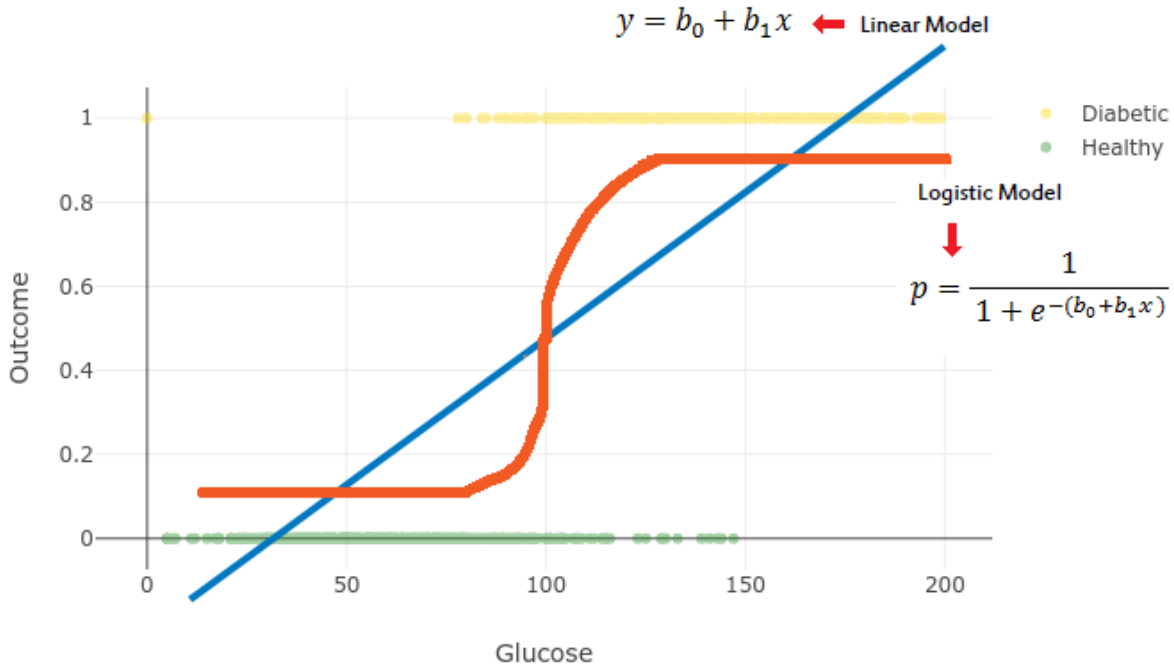
are answerable using Logistic Regression if sufficient data is available and you're lucky enough to believe there are answers to all of these?

But I digress, let's have a look at the mathematical formulation of the Logistic Regression. First, let's start with the Linear Model[33]:

$$y = b_1 x + b_0$$

where x is the data we're going to use to train our model, b_1 controls the slope and b_0 the interception point with the y axis.

[30] https://js.tensorflow.org/api/latest/#tf.data.Dataset.take
[31] https://js.tensorflow.org/api/latest/#tf.data.Dataset.skip
[32] https://js.tensorflow.org/api/latest/#tf.data.Dataset.batch
[33] https://en.wikipedia.org/wiki/Linear_model



We're going to use the softmax[34] function to get probabilities out of the Linear Model and obtain a generalized model of Logistic Regression. Softmax Regression[35] allows us to create a model with more than 2 output classes (binary response):

$$p = \frac{1}{1 + \exp^{-(b_1 x + b_0)}}$$

where b_1 defines the steepness of the curve and b_0 moves the curve left and right.

We want to use our data X and some training magic to learn the parameters b_1 and b_0. Let's use TensorFlow.js for that!

Predicting diabetes

Let's put the theory into practice by building a model into TensorFlow.js and predict the outcome for a patient.

[34] https://en.wikipedia.org/wiki/Softmax_function
[35] http://deeplearning.stanford.edu/tutorial/supervised/SoftmaxRegression/

The model

Remember that the key to building a Logistic Regression model was the Linear Model and applying a softmax function to it:

```
const model = tf.sequential();
model.add(
  tf.layers.dense({
    units: 2,
    activation: "softmax",
    inputShape: [featureCount]
  })
);
```

Note that we have 2 outputs because of the one-hot encoding and dynamic input count, based on the features we've chosen to train the model. Yes, it is that easy to build a Logistic Regression model in TensorFlow.js.

The next step is to compile[36] the model:

```
const optimizer = tf.train.adam(0.001);
model.compile({
  optimizer: optimizer,
  loss: "binaryCrossentropy",
  metrics: ["accuracy"]
});
```

The training process of our model consists of minimizing the loss function. This gets done by the Adam[37] optimizer we're providing. Note that we're providing a learning rate of `0.001`.

The learning rate is known as a hyperparameter since it is a parameter you provide for your model to use. It controls how much each new update should "override" what your model already knows. Choosing the "correct" learning rate is somewhat of voodoo magic.

We're using Cross-Entropy loss[38] (known as log loss) to evaluate how well our model is doing. It (harshly) penalizes wrong answers given from classification models, based on the probabilities they give for each class. Here is the definition:

$$\text{Cross-Entropy} = -\sum_{c=1}^{C} y_{o,c} \log(p_{o,c})$$

[36] https://js.tensorflow.org/api/latest/#tf.LayersModel.compile
[37] https://js.tensorflow.org/api/latest/#train.adam
[38] https://en.wikipedia.org/wiki/Cross_entropy

where C is the number of classes, y is a binary indicator if the class label is the correct classification for the observation and p is the predicted probability that o is of class c.

Note that we request from TensorFlow to record the accuracy metrics.

Training

Let's use fitDataset[39] to train our model using the training and validation datasets we've prepared:

```
const trainLogs = [];
const lossContainer = document.getElementById("loss-cont");
const accContainer = document.getElementById("acc-cont");

await model.fitDataset(trainDs, {
  epochs: 100,
  validationData: validDs,
  callbacks: {
    onEpochEnd: async (epoch, logs) => {
      trainLogs.push(logs);
      tfvis.show.history(lossContainer, trainLogs, ["loss", "val_loss"]);
      tfvis.show.history(accContainer, trainLogs, ["acc", "val_acc"]);
    }
  }
});
```

We train our model for 100 epochs (number of times the whole training set is shown to the model) and record the training logs for visualization using the onEpochEnd[40] callback.

We're going to wrap all of this into a function called `trainLogisticRegression` which is defined as:

```
const trainLogisticRegression = async (
  featureCount,
  trainDs,
  validDs
) => {
...
}
```

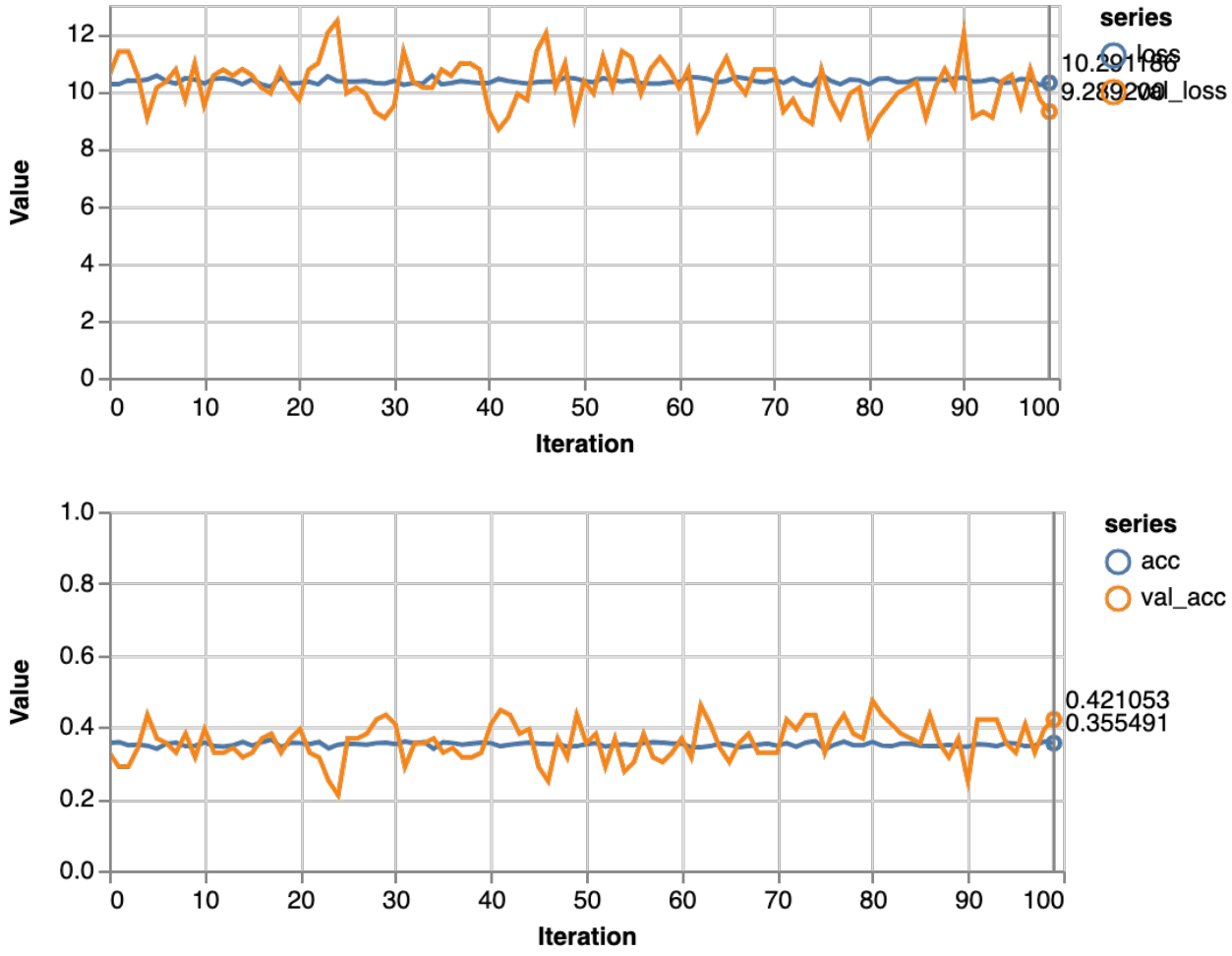
Evaluation

Let's use everything we've built so far to evaluate how well our model is doing:

[39] https://js.tensorflow.org/api/latest/#tf.Sequential.fitDataset
[40] https://js.tensorflow.org/api/latest/#tf.LayersModel.fitDataset

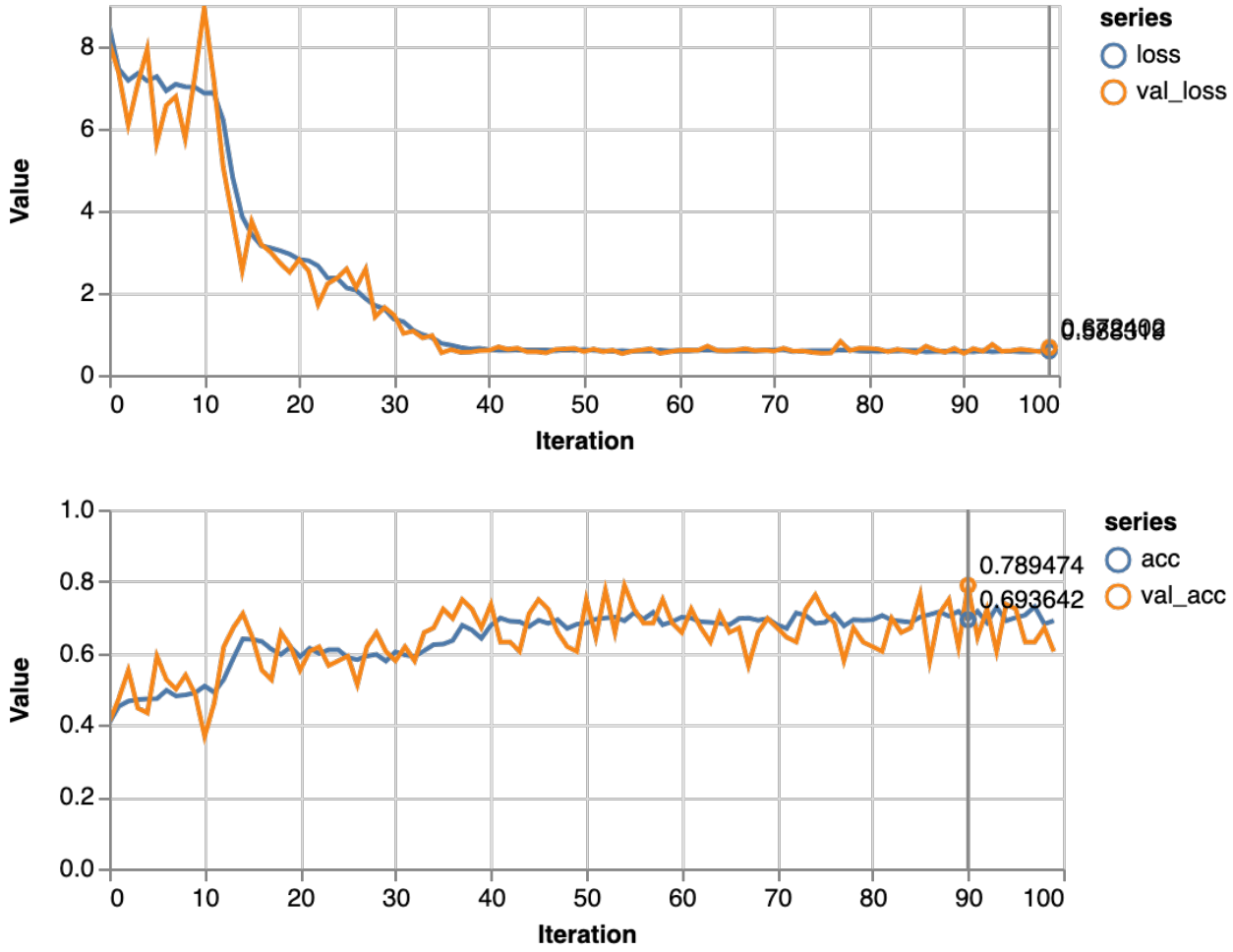
```
const features = ["Glucose"];

const [trainDs, validDs, xTest, yTest] = createDataSets(
  data,
  features,
  0.1,
  16
);

trainLogisticRegression(features.length, trainDs, validDs);
```

Note that we only use the glucose levels for training our model. Here are the results:



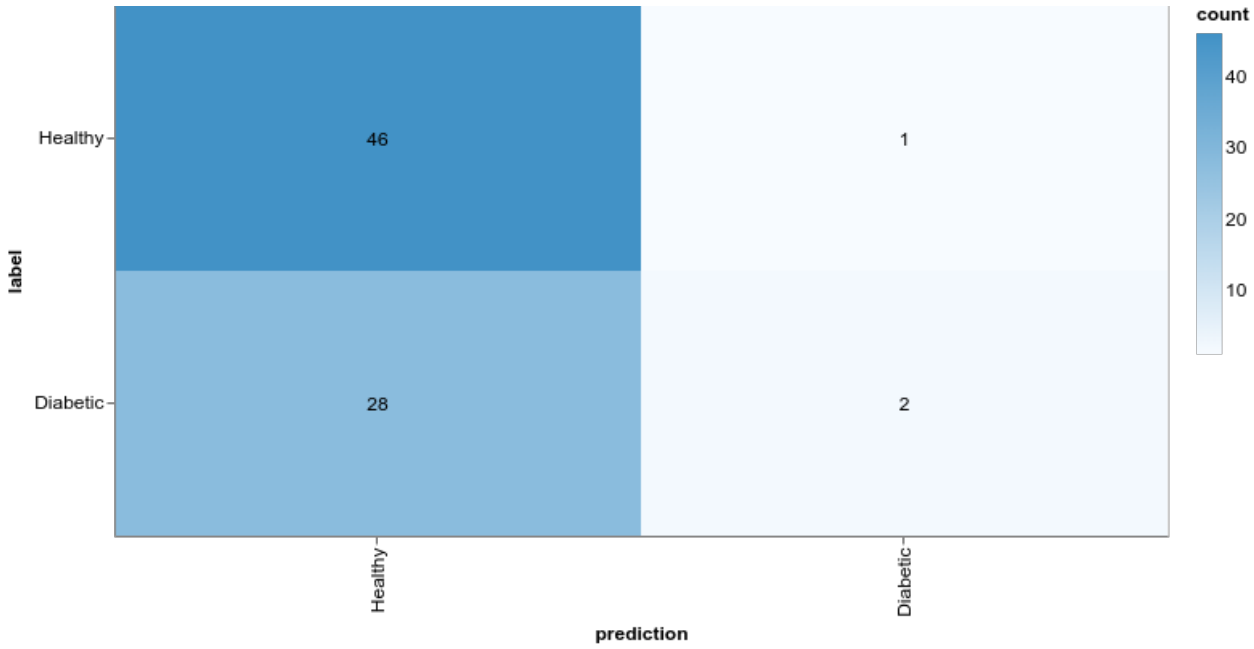
Not good at all. Our model performs worse than a dummy that predicts healthy 65% of the time. Also, the loss never really starts dropping. Let's try with more data:

```
const features = ["Glucose", "Age", "Insulin", "BloodPressure"];

const [trainDs, validDs, xTest, yTest] = createDataSets(
  data,
  features,
  0.1,
  16
);

const model = await trainLogisticRegression(features.length, trainDs, validDs);
```



Much better, the loss value is reduced significantly during training, and we obtain about 79% accuracy on the validation set. Let's take a closer look at the classification performance with a confusion matrix[41]:

[41] https://en.wikipedia.org/wiki/Confusion_matrix



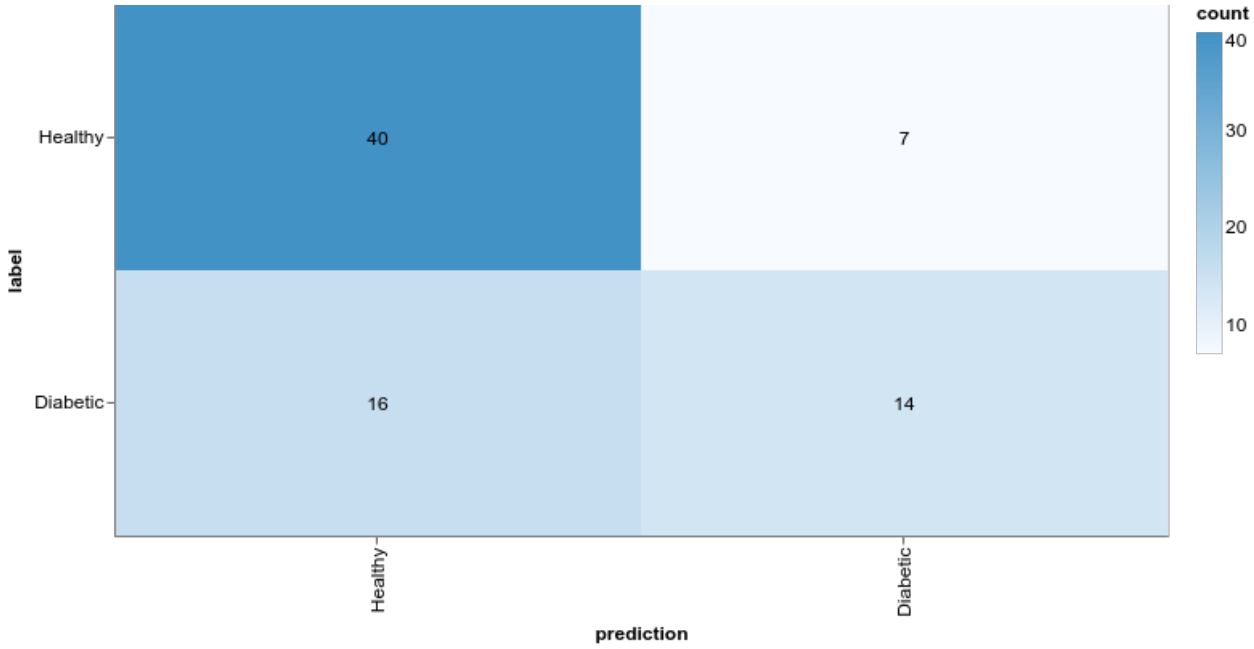
The confusion matrix can be obtained using the model predictions and test set:

```
const preds = model.predict(xTest).argMax(-1);
const labels = yTest.argMax(-1);

const confusionMatrix = await tfvis.metrics.confusionMatrix(labels, preds);

const container = document.getElementById("confusion-matrix");

tfvis.render.confusionMatrix(container, {
  values: confusionMatrix,
  tickLabels: ["Healthy", "Diabetic"]
});
```

Even though our model might've obtained better accuracy, the results are still horrible. Being healthy is vastly overpredicted compared to having diabetes. What if we try with a more complex model:

```
const model = tf.sequential();

model.add(
  tf.layers.dense({
    units: 12,
    activation: "relu",
    inputShape: [featureCount]
  })
);

model.add(
  tf.layers.dense({
    units: 2,
    activation: "softmax"
  })
);
```

Here is the confusion matrix for this model:



We'll not look into this model for now, but note that we obtain much better results by increasing the complexity of the model.

Conclusion

Congratulations! You built and trained not one, but a couple of models, including Logistic Regression, that predicts whether or not a patient has Diabetes. You've also met the real-world - processing data, training and building models, are hard things to do. Moreover, not everything is predictable, no matter how many data points you have.

Run the complete source code for this tutorial right in your browser[42]

That said, there are ways to improve the process of building and training models. We know that using some techniques is better than others, in a certain context. Well, Machine Learning is nuanced :)

References

- Logistic Regression by Dr. Saed Sayad[43]
- A Gentle Introduction to TensorFlow.js[44]

[42] https://codesandbox.io/s/logistic-regression-tensorflow-js-r6b5m?fontsize=14
[43] https://www.saedsayad.com/logistic_regression.htm
[44] https://medium.com/tensorflow/a-gentle-introduction-to-tensorflow-js-dba2e5257702

House Price Prediction using Linear Regression

> TL;DR Build a Linear Regression model in TensorFlow.js to predict house prices. Learn how to handle categorical data and do feature scaling.

Raining again. It has been 3 weeks since the last time you saw the sun. You're getting tired of all this cold and unpleasant feeling of loneliness and melancholy.

The voice in your head is getting louder and louder.

"MOVE".

Alright, you're ready to do it. Where to? You remember that you're nearly broke.

A friend of yours told you about this place Ames, Iowa and it stuck in your head. After a quick search, you found that the weather is pleasant during the year and there is some rain, but not much. Excitement!

Fortunately, you know of this dataset on Kaggle that might help you find out how much your dream house might cost. Let's get to it!

Run the complete source code for this tutorial right in your browser[45]

House prices data

Our data comes from Kaggle's House Prices: Advanced Regression Techniques[46] challenge.

> With 79 explanatory variables describing (almost) every aspect of residential homes in Ames, Iowa, this competition challenges you to predict the final price of each home.

Here's a subset of the data we're going to use for our model:

- `OverallQual` - Rates the overall material and finish of the house (0 - 10)
- `GrLivArea` - Above grade (ground) living area square feet
- `GarageCars` - Size of garage in car capacity
- `TotalBsmtSF` - Total square feet of basement area
- `FullBath` - Full bathrooms above grade

[45] https://codesandbox.io/s/logistic-regression-with-tensorflow-js-v95ur?fontsize=14
[46] https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data

- `YearBuilt` - Original construction date
- `SalePrice` - The property's sale price in dollars (we're trying to predict this)

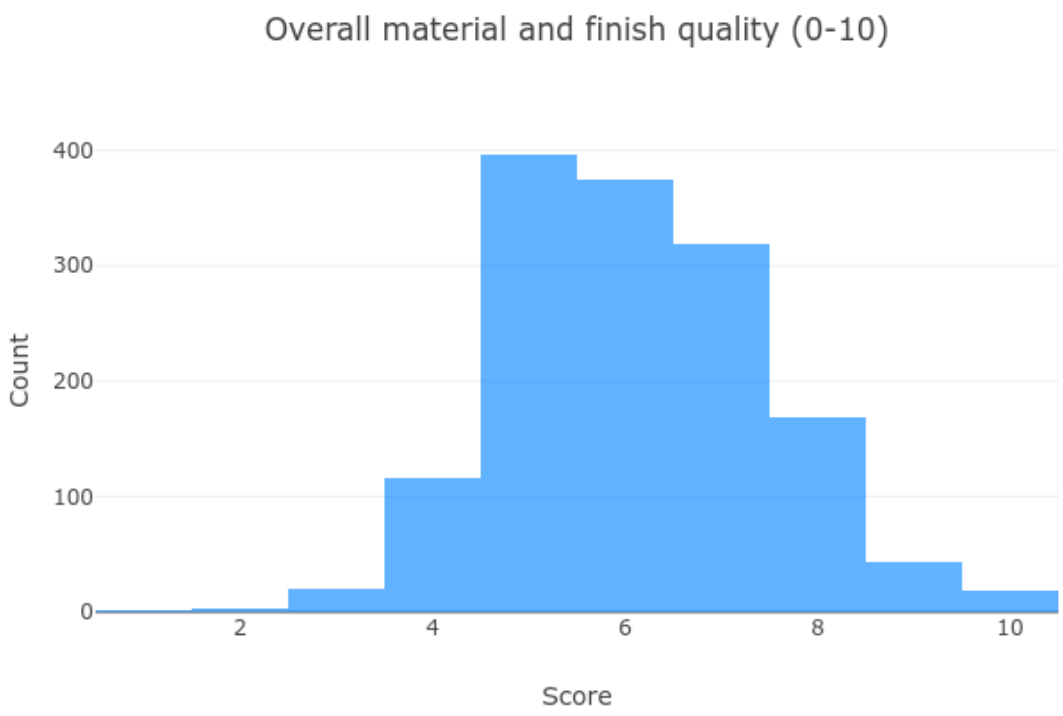
Let's use Papa Parse[47] to load the training data:

```
const prepareData = async () => {
  const csv = await Papa.parsePromise(
    "https://raw.githubusercontent.com/curiousily/Linear-Regression-with-TensorFlow-\
js/master/src/data/housing.csv"
  );

  return csv.data;
};
```

```
const data = await prepareData();
```

Exploration

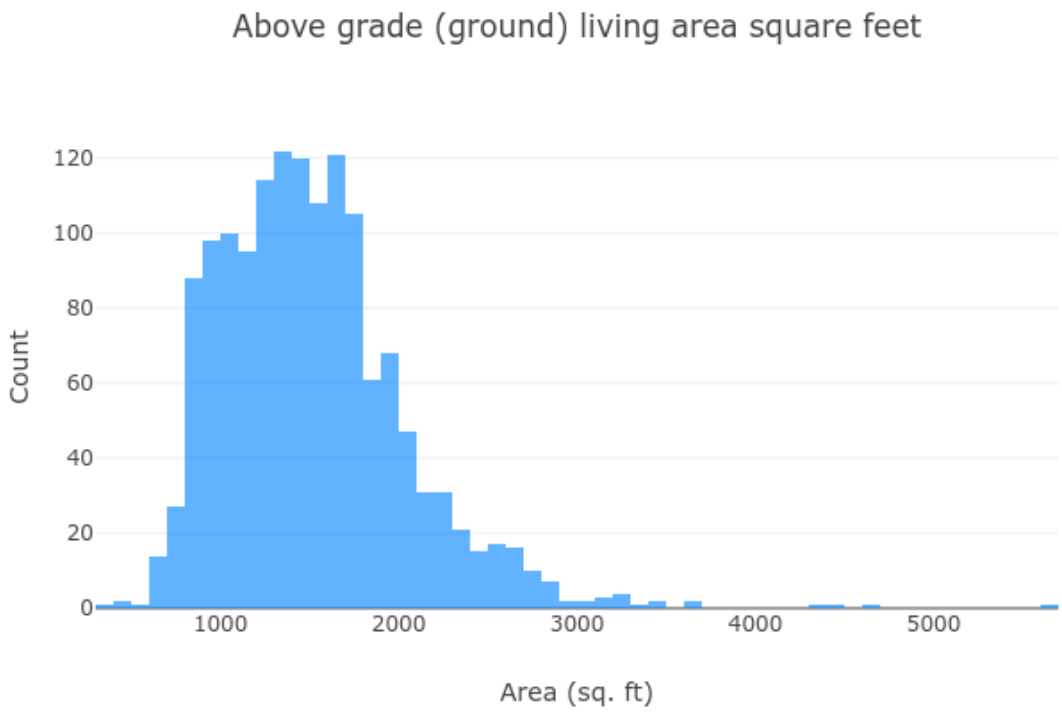
Let's build a better understanding of our data. First - the quality score of each house:



[47] https://www.papaparse.com/

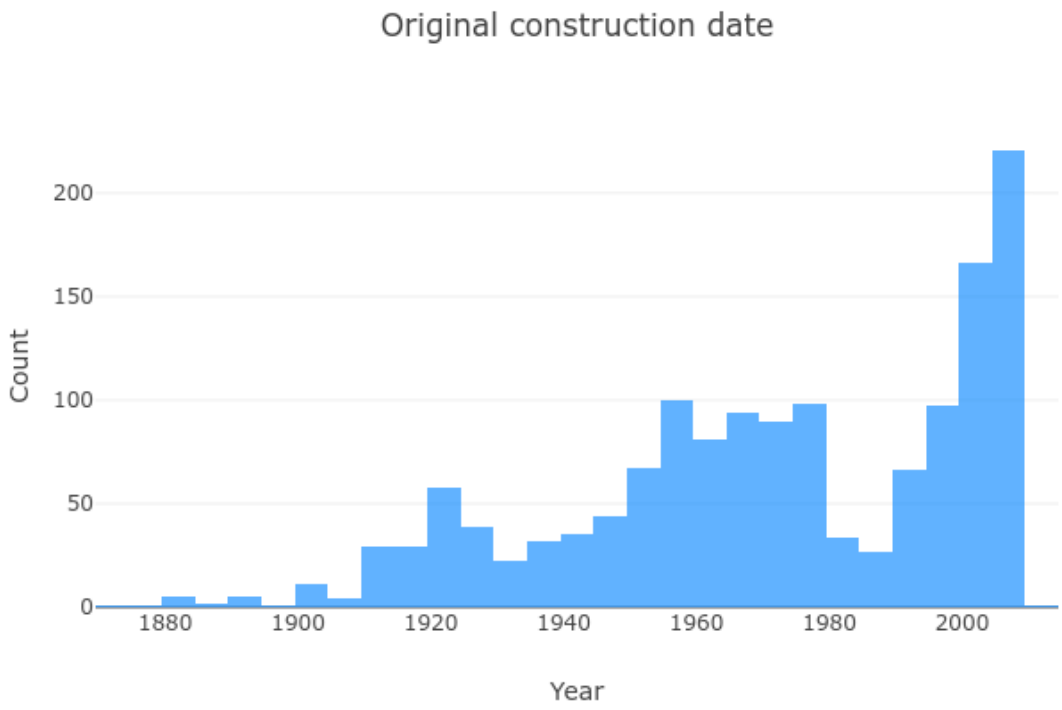
Most houses are of average quality, but there are more “good” than “bad” ones.

Let’s see how large are they (that’s what she said):



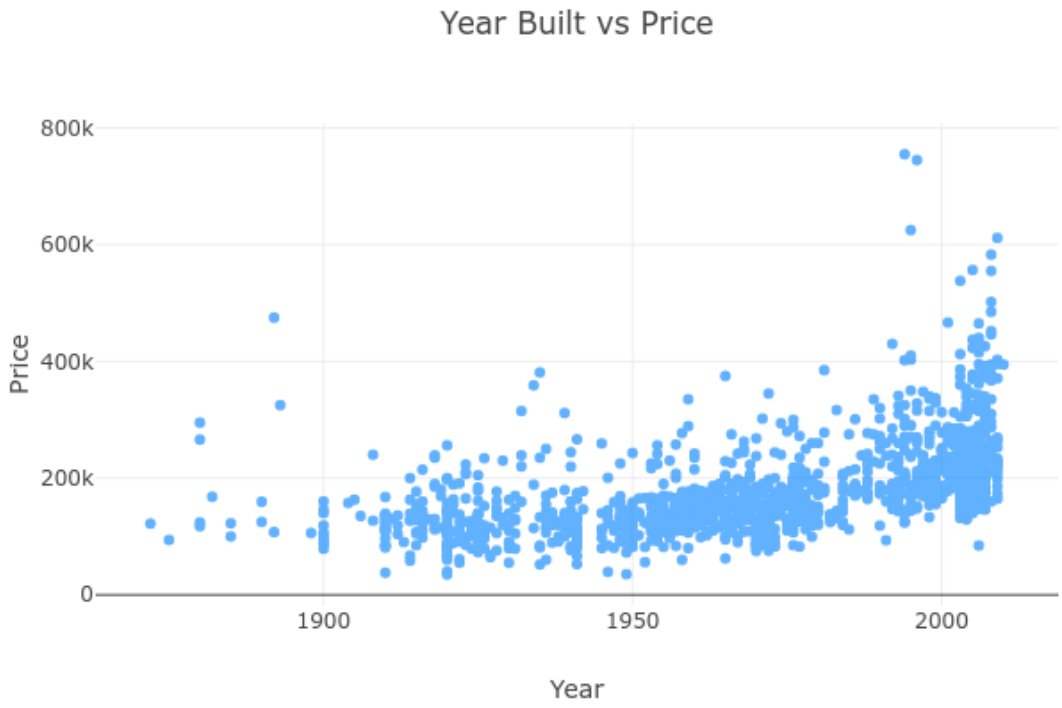
Most of the houses are within the 1,000 - 2,000 range, and we have some that are bigger.

Let’s have a look at the year they are built:



Even though there are a lot of houses that were built recently, we have a much more widespread distribution.

How related is the year with the price?



Seems like newer houses are pricier, no love for the old and well made then?

Oh ok, but higher quality should equal higher price, right?

Generally yes, but look at quality 10. Some of those are relatively cheap. Any ideas why that might be?

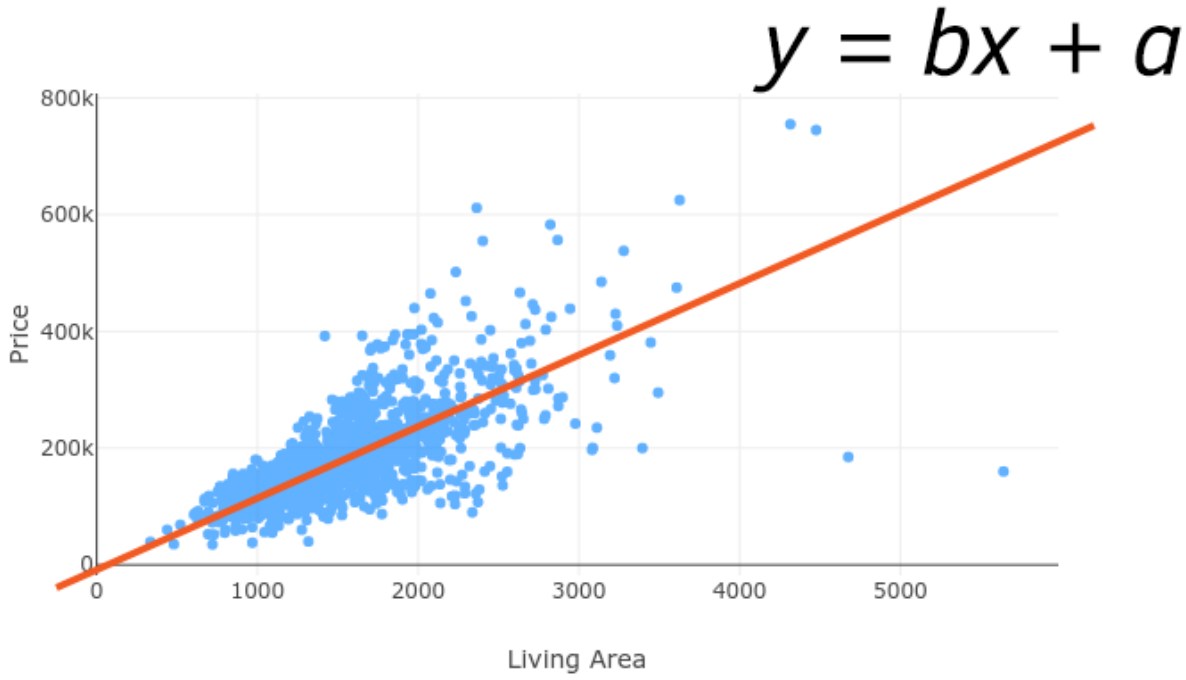
Is a larger house equal higher price?

Seems like it, we might start our price prediction model using the living area!

Linear Regression

Linear Regression models assume that there is a linear relationship (can be modeled using a straight line) between a dependent continuous variable `Y` and one or more explanatory (independent) variables `X`.

In our case, we're going to use features like `living area` (X) to predict the `sale price` (Y) of a house.



Simple Linear Regression

Simple Linear Regression is a model that has a single independent variable `X`. It is given by:

$$Y = bX + a$$

Where a and b are parameters, learned during the training of our model. X is the data we're going to use to train our model, b controls the slope and a the interception point with the y axis.

Multiple Linear Regression

A natural extension of the Simple Linear Regression model is the multivariate one. It is given by:

$$Y(x_1, x_2, \ldots, x_n) = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + w_0$$

where $x_1, x_2 \ldots, x_n$ are features from our dataset and $w_1, w_2 \ldots, w_n$ are learned parameters.

Loss function

We're going to use Root Mean Squared Error[48] to measure how far our predictions are from the real house prices. It is given by:

$$RMSE = J(W) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - h_w(x^{(i)}))^2}$$

where the hypothesis/prediction h_w is given by:

$$h_w(x) = g(w^T x)$$

Data Preprocessing

Currently, our data sits into an array of JS objects. We need to turn it into Tensors[49] and use it for training our model(s). Here is the code for that:

```
const createDataSets = (data, features, categoricalFeatures, testSize) => {
  const X = data.map(r =>
    features.flatMap(f => {
      if (categoricalFeatures.has(f)) {
        return oneHot(!r[f] ? 0 : r[f], VARIABLE_CATEGORY_COUNT[f]);
      }
      return !r[f] ? 0 : r[f];
    })
  );

  const X_t = normalize(tf.tensor2d(X));

  const y = tf.tensor(data.map(r => (!r.SalePrice ? 0 : r.SalePrice)));

  const splitIdx = parseInt((1 - testSize) * data.length, 10);

  const [xTrain, xTest] = tf.split(X_t, [splitIdx, data.length - splitIdx]);
  const [yTrain, yTest] = tf.split(y, [splitIdx, data.length - splitIdx]);

  return [xTrain, xTest, yTrain, yTest];
};
```

[48] https://en.wikipedia.org/wiki/Root-mean-square_deviation
[49] https://js.tensorflow.org/api/latest/#Tensors

We store our features in `X` and the labels in `y`. Then we convert the data into Tensors and split it into training and testing datasets.

Categorical features

Some of the features in our dataset are categorical/enumerable. For example, `GarageCars` can be in the 0-5 range.

Leaving categories represented as integers in our dataset might introduce implicit ordering dependence. Something that does not exist with categorical variables.

We'll use one-hot encoding[50] from TensorFlow to create an integer vector for each value to break the ordering. First, let's specify how many different values each category has:

```
const VARIABLE_CATEGORY_COUNT = {
  OverallQual: 10,
  GarageCars: 5,
  FullBath: 4
};
```

We'll use tf.oneHot()[51] to convert individual value to a one-hot representation:

```
const oneHot = (val, categoryCount) =>
  Array.from(tf.oneHot(val, categoryCount).dataSync());
```

Note that the `createDataSets()` function accepts a parameter called `categoricalFeatures` which should be a set. We'll use this to check whether or not we should process this feature as categorical.

Feature scaling

Feature scaling[52] is used to transform the feature values into a (similar) range. Feature scaling will help our model(s) learn faster[53] since we're using Gradient Descent[54] for training it.

Let's use one of the simplest method for feature scaling - min-max normalization:

[50] https://en.wikipedia.org/wiki/One-hot
[51] https://js.tensorflow.org/api/latest/#oneHot
[52] https://en.wikipedia.org/wiki/Feature_scaling
[53] https://arxiv.org/abs/1502.03167
[54] https://en.wikipedia.org/wiki/Gradient_descent

```
const normalize = tensor =>
  tf.div(
    tf.sub(tensor, tf.min(tensor)),
    tf.sub(tf.max(tensor), tf.min(tensor))
  );
```

this method rescales the range of values in the range of [0, 1].

Predicting house prices

Now that we know about the Linear Regression model(s), we can try to predict house prices based on the data we have. Let's start simple:

Building a Simple Linear Regression model

We'll wrap the training process in a function that we can reuse for our future model(s):

```
const trainLinearModel = async (xTrain, yTrain) => {
  ...
}
```

`trainLinearModel` accepts the features and labels for our model. Let's define a Linear Regression model using TensorFlow:

```
const model = tf.sequential();

model.add(
  tf.layers.dense({
    inputShape: [xTrain.shape[1]],
    units: xTrain.shape[1]
  })
);

model.add(tf.layers.dense({ units: 1 }));
```

Since TensorFlow.js doesn't offer RMSE loss function, we'll use MSE and take the square root of that later. We'll also track Mean Absolute Error (MAE) between the predictions and real prices:

```
model.compile({
  optimizer: tf.train.sgd(0.001),
  loss: "meanSquaredError",
  metrics: [tf.metrics.meanAbsoluteError]
});
```

Here's the training process:

```
const trainLogs = [];
const lossContainer = document.getElementById("loss-cont");
const accContainer = document.getElementById("acc-cont");

await model.fit(xTrain, yTrain, {
  batchSize: 32,
  epochs: 100,
  shuffle: true,
  validationSplit: 0.1,
  callbacks: {
    onEpochEnd: async (epoch, logs) => {
      trainLogs.push({
        rmse: Math.sqrt(logs.loss),
        val_rmse: Math.sqrt(logs.val_loss),
        mae: logs.meanAbsoluteError,
        val_mae: logs.val_meanAbsoluteError
      });
      tfvis.show.history(lossContainer, trainLogs, ["rmse", "val_rmse"]);
      tfvis.show.history(accContainer, trainLogs, ["mae", "val_mae"]);
    }
  }
});
```

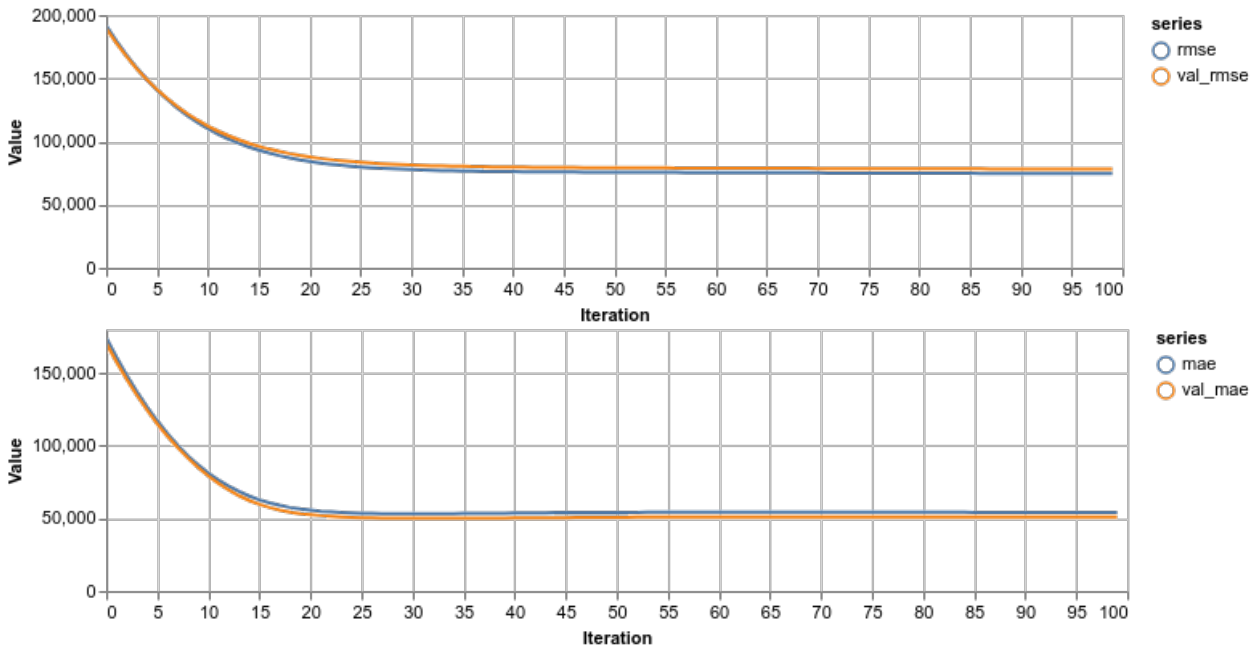
We train for *100* epochs, shuffle the data beforehand, and use *10%* of it for validation. The RMSE and MAE are visualized after each epoch.

Training

Our Simple Linear Regression model is using the *GrLivArea* feature:

```
const [xTrainSimple, xTestSimple, yTrainSimple, yTestIgnored] = createDataSets(
  data,
  ["GrLivArea"],
  new Set(),
  0.1
);

const simpleLinearModel = await trainLinearModel(xTrainSimple, yTrainSimple);
```

We don't have categorical features, so we leave that set is empty. Let's have a look at the performance:



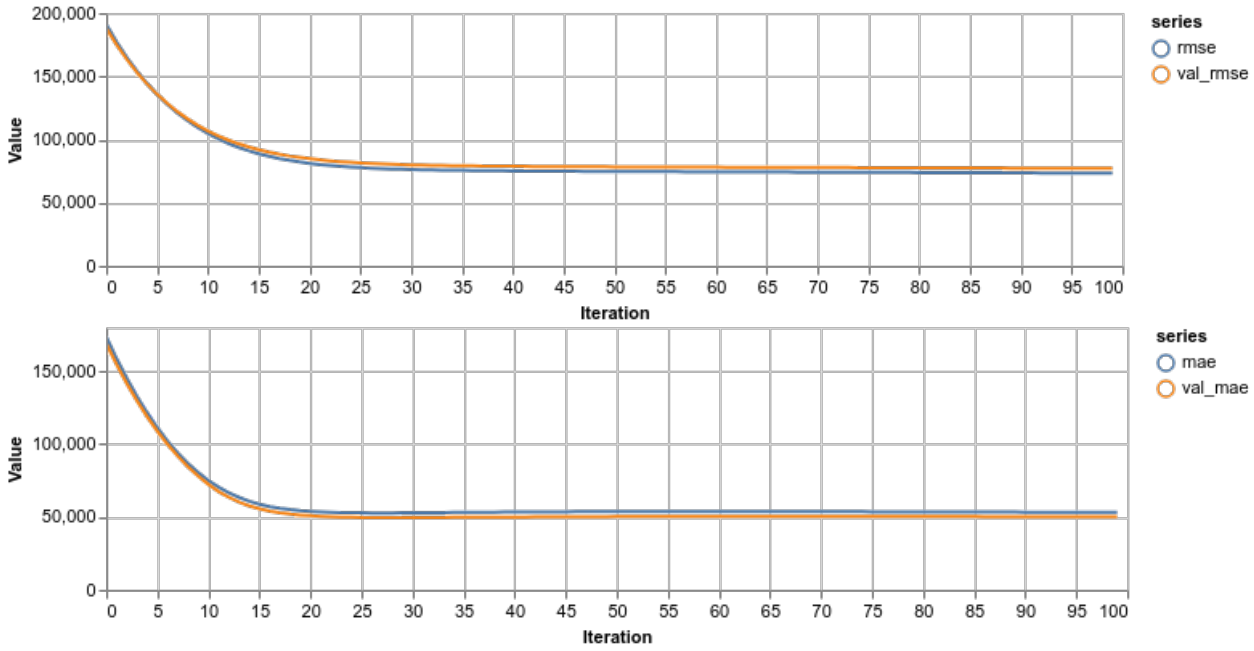
Building a Multiple Linear Regression model

We have a lot more data we haven't used yet. Let's see if that will help improve the predictions:

```
const features = [
  "OverallQual",
  "GrLivArea",
  "GarageCars",
  "TotalBsmtSF",
  "FullBath",
  "YearBuilt"
];

const categoricalFeatures = new Set(["OverallQual", "GarageCars", "FullBath"]);
```

```
11
12 const [xTrain, xTest, yTrain, yTest] = createDataSets(
13   data,
14   features,
15   categoricalFeatures,
16   0.1
17 );
```

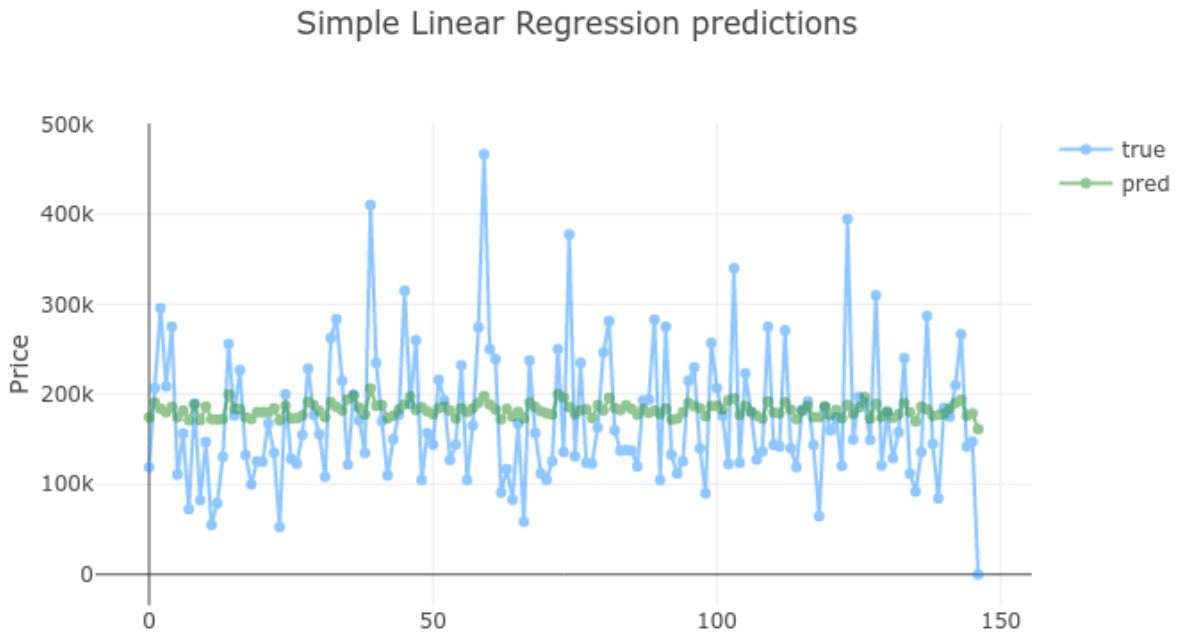
We use all features in our dataset and pass a set of the categorical ones. Did we do better?



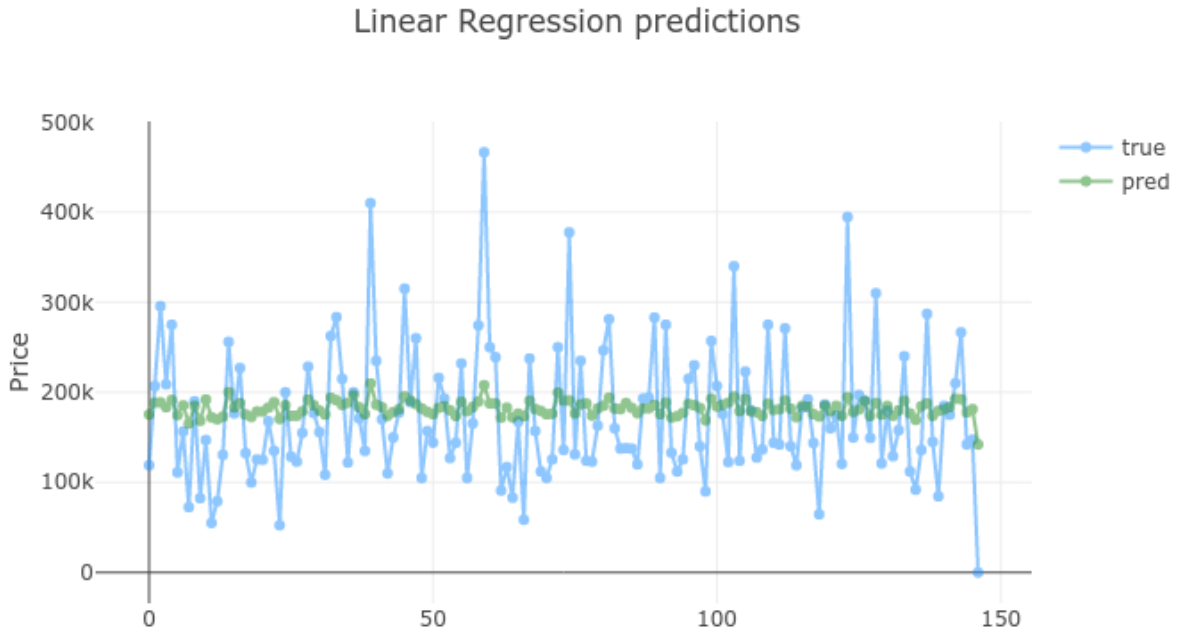
Overall, both models are performing at about the same level. This time, increasing the model complexity didn't give us better accuracy.

Evaluation

Another way to evaluate our models is to check their predictions against the test data. Let's start with the Simple Linear Regression:



How did adding more data improved the predictions?



Well, it didn't. Again, having a more complex model trained with more data didn't provide better performance.

Conclusion

You did it! You built two Linear Regression models that predict house price based on a set of features. You also did:

- Feature scaling for faster model training
- Convert categorical variables into one-hot representations
- Implement RMSE (based on MSE) for accuracy evaluation

Run the complete source code for this tutorial right in your browser[55]

Is it time to learn about Neural Networks?

References

Handling Categorical Data in Machine Learning Models[56]

[55] https://codesandbox.io/s/logistic-regression-with-tensorflow-js-v95ur?fontsize=14
[56] https://www.pluralsight.com/guides/handling-categorical-data-in-machine-learning-models

About Feature Scaling and Normalization[57]

RMSE: Root Mean Square Error[58]

[57] https://sebastianraschka.com/Articles/2014_about_feature_scaling.html
[58] https://www.statisticshowto.datasciencecentral.com/rmse/

Build a simple Neural Network

> TL;DR Build a simple Neural Network model in TensorFlow.js to make a laptop buying decision. Learn why Neural Networks need activation functions and how should you initialize their weights.

It is in the middle night, and you're dreaming some rather alarming dreams with a smile on your face. Suddenly, your phone starts ringing, rather internationally. You pick up, half-asleep, and listen to something bizarre.

A friend of yours is calling, from the other side of our planet, asking for help in picking a laptop. After all, it is Black Friday!

You're a bit dazzled by the fact that this is the first time you hear from your friend in 5 years. Still, you're a good person and agree to help out. Maybe it is time to put your TensorFlow.js skills into practice?

How about you build a model to help out your friend so you can get back to sleep? You heard that Neural Networks are pretty hot right now. It is 3 in the morning, there isn't much need for persuasion in your mind. You'll use a Neural Network for this one!

Run the complete source code for this tutorial right in your browser[59]

Neural Networks

What is a Neural Network? In a classical cliff-hanger fashion, we'll start far away from answering this question.

Neural Networks were around for a while (since 1950s)? Why did they become popular just recently (last 5-10 years)? First introduced by Warren McCulloch and Walter Pitts in A logical calculus of the ideas immanent in nervous activity[60] Neural Networks were really popular until the mid-1980s when Support Vector Machines[61] and other methods overtook the community.

The Universal approximation theorem[62] states that a Neural Networks can approximate any function (under some mild assumptions), even with a single hidden layer (more on that later). One of the first proves was done by George Cybenko[63] in 1989 for sigmoid activation functions[64] (will have a look at those in a bit).

[59] https://codesandbox.io/s/simple-neural-network-with-tensorflow-js-7k0jr?fontsize=14
[60] https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf
[61] https://en.wikipedia.org/wiki/Support_vector_machine
[62] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.2647&rep=rep1&type=pdf
[63] https://en.wikipedia.org/wiki/George_Cybenko
[64] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.441.7873&rep=rep1&type=pdf

More recently, more and more advances in the field of Deep Learning made Neural Networks a hot topic again. Why? We'll discuss that a bit later. First, let's start with the basics!

The Perceptron

The original model, intended to model how the human brain processed visual data and learned to recognize objects, was susggested by Frank Rosenblatt[65] in the 1950s. The Perceptron takes one or more binary inputs $x_1, x_2, \ldots, x_n$ and produces a binary output:

To compute the output you have to:

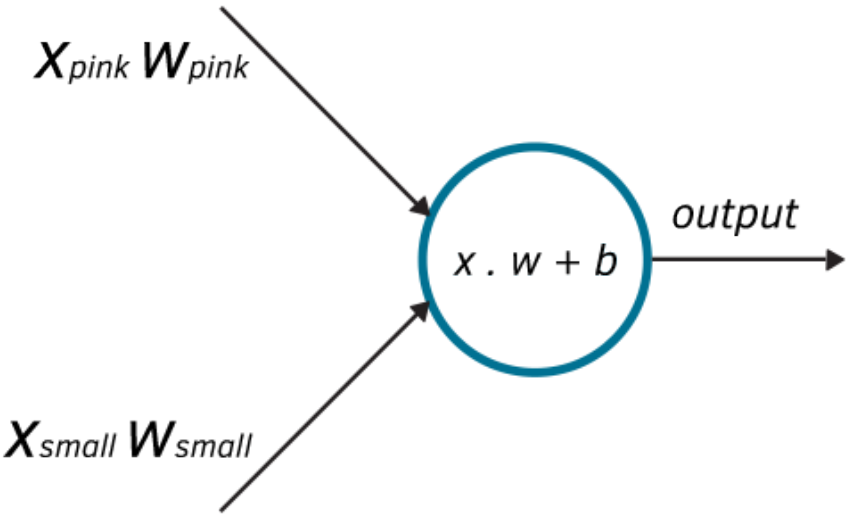
- have weights $w_1, w_2, \ldots, w_n$ expressing the importance of the respective input
- the binary output (0 or 1) is determined by whether the weighted sum $\sum_j w_j x_j$ is greater or lower than some threshold

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j < \text{threshold} \\ 1 & \text{otherwise} \end{cases}$$

Let's have a look at an example. Imagine you need to decide whether or not you need a new laptop. The most important features are its color and size (that's what she said). So, you have two inputs:

1. is it pink?
2. is it small (gotcha)?

[65] https://en.wikipedia.org/wiki/Frank_Rosenblatt



You can represent these factors with binary variables x_{pink}, x_{small} and assign weights/importance w_{pink}, w_{small} to each one. Depending on the importance you assign to each factor, you can get different models.

We can simplify the Perceptron even further. We can rewrite $\sum_j w_j x_j$ as a dot product of two vectors $w \cdot x$. Next, we'll introduce the Perceptron's bias, $b = -$threshold. Using it, we can rewrite the model as:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b < 0 \\ 1 & \text{otherwise} \end{cases}$$

The bias is a measure of how easy it is for a perceptron to output 1 (to fire). Large positive bias makes outputting 1 easy, while a large negative bias makes it difficult.

Let's build the Perceptron model using TensorFlow.js:

```
const perceptron = ({ x, w, bias }) => {
  const product = tf.dot(x, w).dataSync()[0];
  return product + bias < 0 ? 0 : 1;
};
```

An offer for a laptop comes around. It is not pink, but it is small $x = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. You're biased towards not buying a laptop because you're broke. You can encode that with a negative bias. You're one of the brainier users, and you put more emphasis on size, rather than color $w = \begin{bmatrix} 0.5 \\ 0.9 \end{bmatrix}$:

```
perceptron({
  x: [0, 1],
  w: [0.5, 0.9],
  bias: -0.5
});
```

```
1
```

Yes, you have to buy that laptop!

Sigmoid neuron

To make learning from data possible, we want the weights of our model to change only by a small amount when presented with an example. That is, each example should cause a small change in the output.

That way, one can continuously adjust the weights while presenting new data and not worrying that a single example will wipe out everything the model has learned so far.

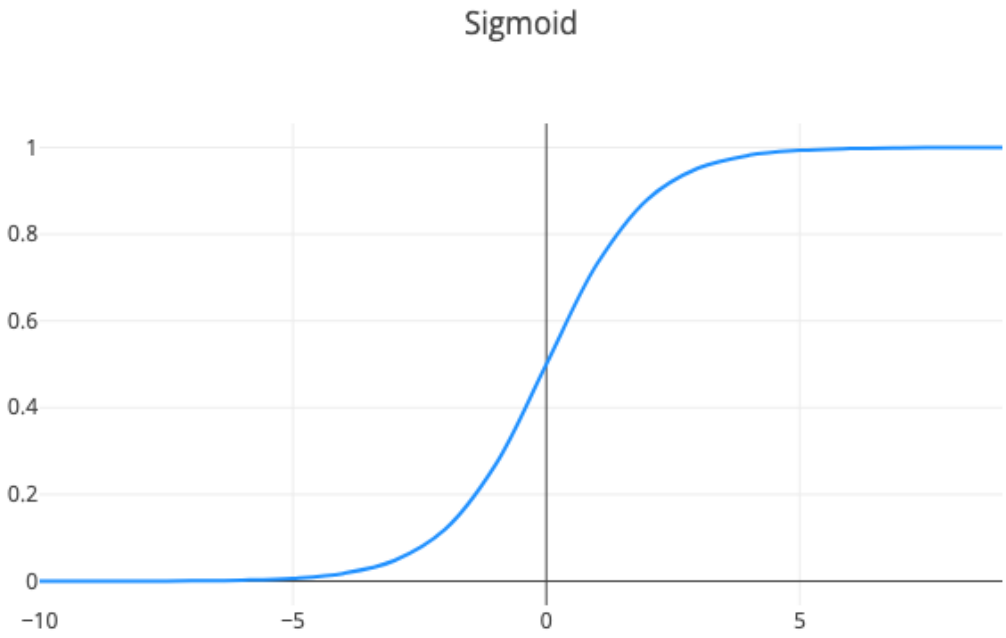
The Perceptron is not an ideal for that purpose since small changes in the inputs are propagated linearly to the output. We can overcome this using a *sigmoid* neuron.

The sigmoid neuron has inputs $x_1, x_2, \ldots, x_n$ that can be values between 0 and 1. The output is given by $\sigma(w \cdot x + b)$ where σ is the *sigmoid function*, defined by:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Let's have a look at it using TensorFlow.js and Plotly:

```
const xs = [...Array(20).keys()].map(x => x - 10);
const ys = tf.sigmoid(xs).dataSync();

renderActivationFunction(xs, ys, "Sigmoid", "sigmoid-cont");
```

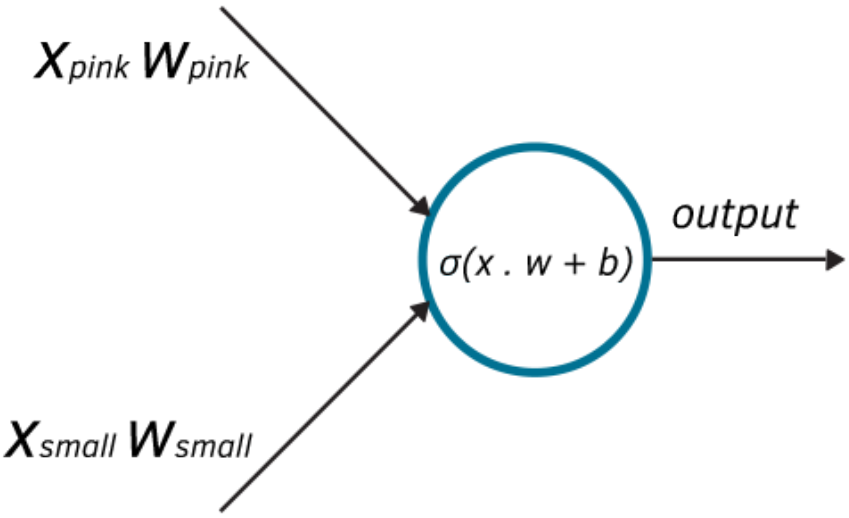


Using the weights and inputs we get:

$$\sigma = \frac{1}{1 + e^{-(\sum_j w_j x_j - b)}}.$$

Let's dive deeper into the sigmoid neuron and understand the similarities with the Perceptron:

- Suppose that z is a *large positive number*. Then $e^{-z} \approx 0$ and $\sigma(z) \approx 1$.
- Suppose that z is a *large negative number*. Then $e^{-z} \to \infty$ and $\sigma(z) \approx 0$.
- When z is somewhat modest, we observe a significant difference compared to the Perceptron.



Let's build the *sigmoid neuron* model using TensorFlow.js:

```
const sigmoidPerceptron = ({ x, w, bias }) => {
  const product = tf.dot(x, w).dataSync()[0];
  return tf.sigmoid(product + bias).dataSync()[0];
};
```

Another offer for a laptop comes around. This time you can specify the degree of how close the color is to pink and how small it is.

The color is somewhat pink, and the size is just about right $x = \begin{bmatrix} 0.6 \\ 0.9 \end{bmatrix}$. The rest stays the same:

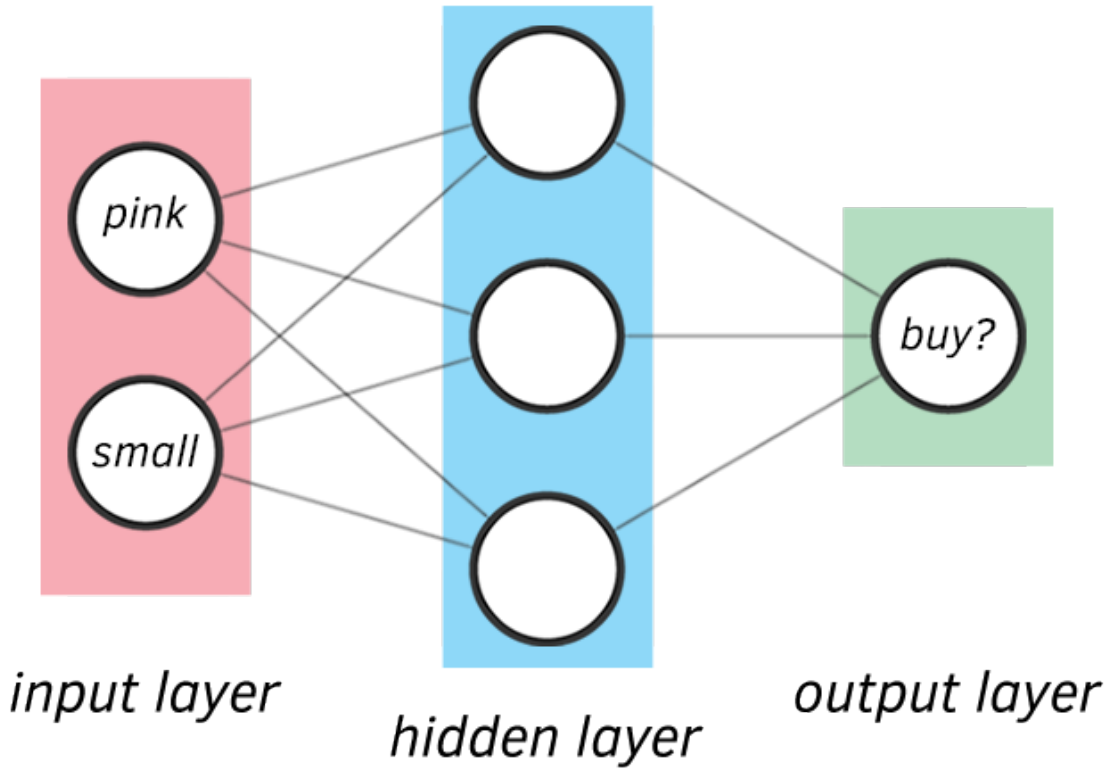
```
sigmoidPerceptron({
  x: [0.6, 0.9],
  w: [0.5, 0.9],
  bias: -0.5
});
```

```
0.6479407548904419
```

Yes, you still want to buy this laptop, but this model also outputs the confidence of its decision. Cool, right?

Architecting Neural Networks

A natural way to extend the models presented above is to group them in some way. One way to do that is to create layers of neurons. Here's a simple Neural Network that can be used to make the decision of buying a laptop:



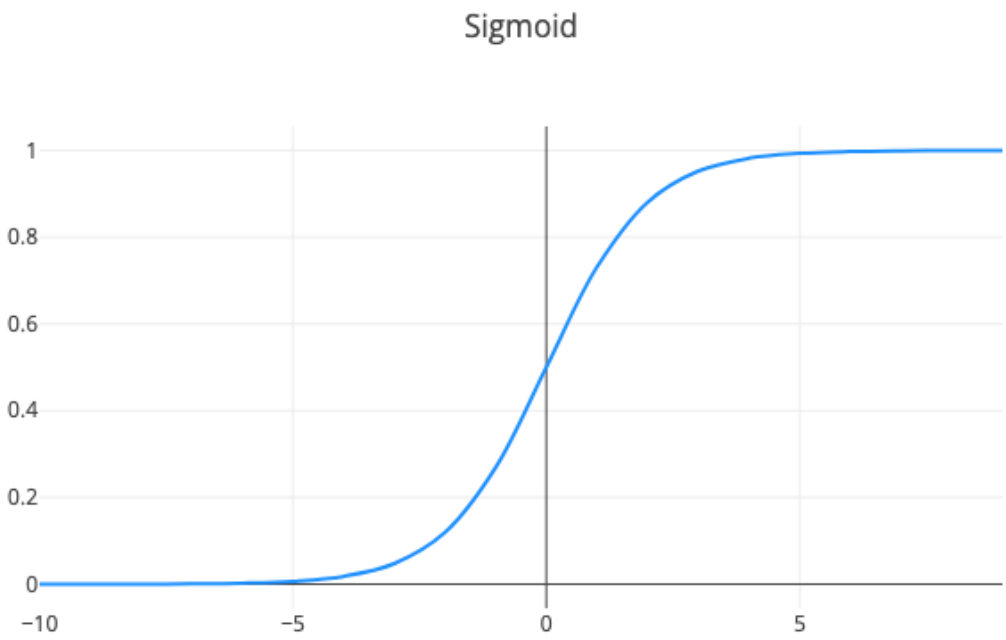
Neural Networks are a collection of neurons, connected in an acyclic graph. Outputs of some neurons are used as inputs to other neurons. They are organized into layers. Our example is composed of fully-connected layers (all neurons between two adjacent layers are connected), and it is a 2 layer Neural Network (we do not count the input layer). Neural Networks can make complex decisions thanks to combination of simple decisions made by the neurons that construct them.

Of course, the output layer contains the answer(s) you're looking for. Let's have a look at some of the ingredients that make training Neural Networks possible:

Activation functions

The Perceptron model is just a linear transformation. Stacking multiple such neurons on each other results in a vector product and a bias addition. Unfortunately, there are a lot of functions that can't be estimated by a linear transformation.

The activation function makes it possible for the model to approximate non-linear functions (predict more complex phenomena). The good thing is, you've already met one activation function - the sigmoid:



One major disadvantage of the Sigmoid function is the is that it becomes really flat outside the [-3, +3] range. This leads to weights getting close to 0 - no learning is happening.

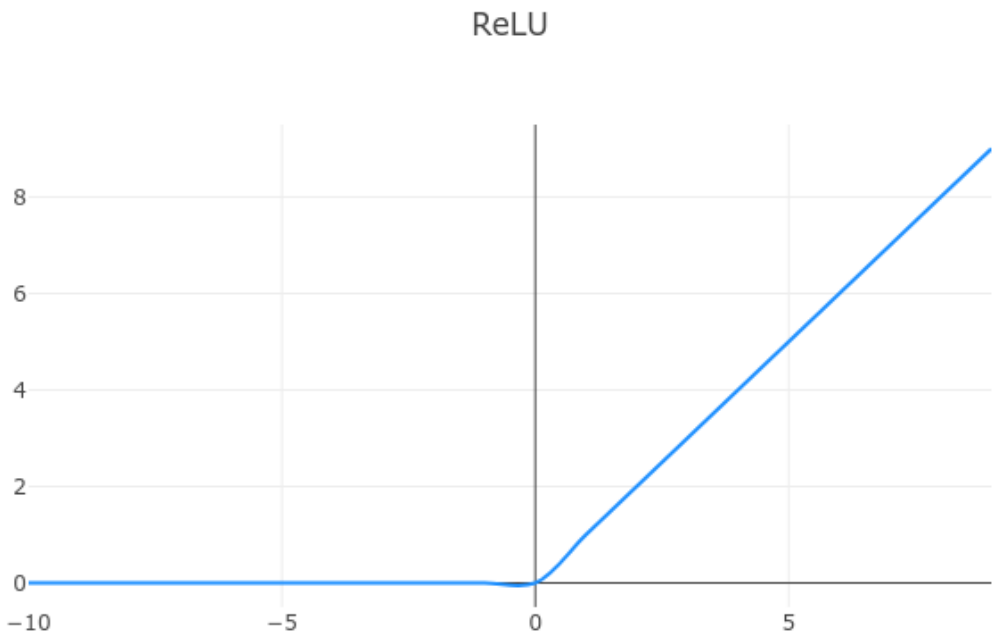
ReLU

ReLU, introduced in the context of Neural Networks in Rectified Linear Units Improve Restricted Boltzmann Machines[66], have a linear output at values greater than 0 and 0 otherwise.

Let's have a look:

[66] https://www.cs.toronto.edu/~hinton/absps/reluICML.pdf

```
const xs = [...Array(20).keys()].map(x => x - 10);
const ys = tf.relu(xs).dataSync();

renderActivationFunction(xs, ys, "ReLU", "relu-cont");
```



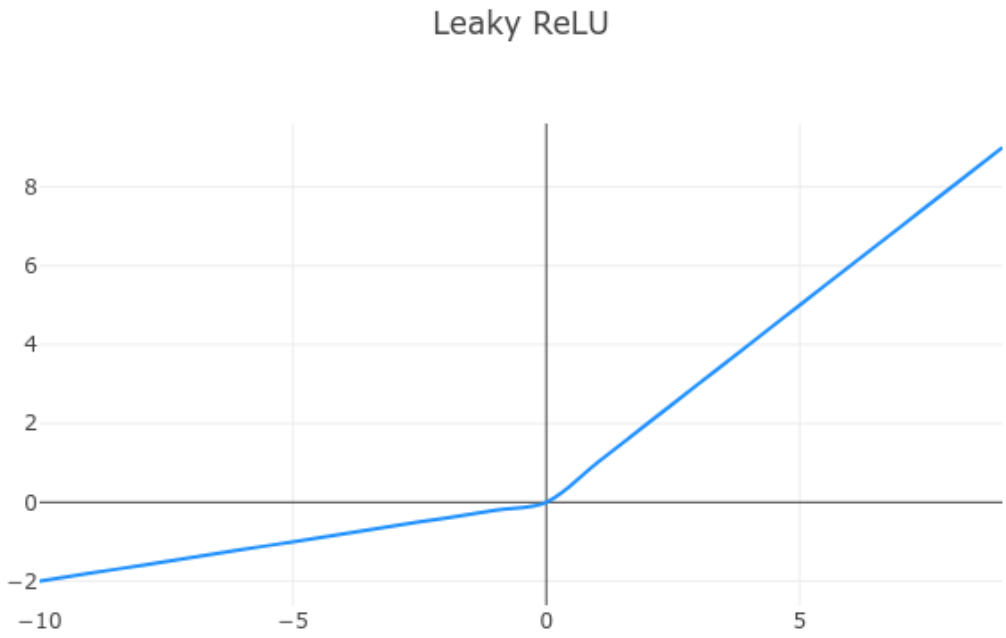
One disadvantage of ReLU is that negative values "die out" and stay at `0` - no learning.

Leaky ReLU

Leaky ReLU, introduced in Rectifier Nonlinearities Improve Neural Network Acoustic Models[67], solves the dead values introduced by ReLu:

```
const xs = [...Array(20).keys()].map(x => x - 10);
const ys = tf.leakyRelu(xs).dataSync();

renderActivationFunction(xs, ys, "Leaky ReLU", "leaky-relu-cont");
```

[67] https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf



Note that negative values get scaled instead of zeroed out. Scaling is adjustable by a parameter in tf.leakyRelu()[68].

Weight initialization

The process of teaching a Neural Network to make "reasonable" predictions involves adjusting the weights of the neurons multiple times. Those weights need to have initial values. How should you choose those?

The initialization process must take into account the algorithm we're using to train our model. More often than not, that algorithm is Stochastic gradient descent (SGD)[69]. Its job is to do a search over possible parameters/weights and choose those that minimize the errors our model makes. Moreover, the algorithm heavily relies on randomness and a good starting point (given by the weights).

Same constant initialization

Imagine that we initialize the weights using the same constant (yes, including 0). Every neuron in the network will compute the same output, which results in the same weight/parameter update. We just defeated the purpose of having multiple neurons.

[68] https://js.tensorflow.org/api/latest/#leakyRelu
[69] https://en.wikipedia.org/wiki/Stochastic_gradient_descent

Too small/large value initialization

Let's initialize the weights with a set of small values. Passing those values to the activation functions will decrease them exponentially, leaving every weight equally unimportant.

On the other hand, initializing with large values will lead to an exponential increase, making the weights equally unimportant again.

Random small number initialization

We can use a Normal distribution[70] with a mean 0 and standard deviation 1 to initialize the weights with small random numbers.

Every neuron will compute different output, which leads to different parameter updates. Of course, multiple other ways exist. Check the TensorFlow.js Initializers[71]

Should you buy the laptop?

Now that you know some Neural Network kung-fu, we can use TensorFlow.js to build a simple model and decide whether you should buy a given laptop.

Laptop data

Let's say that for your friend, size is much more important than the degree of pinkness! You sit down and devise the following dataset:

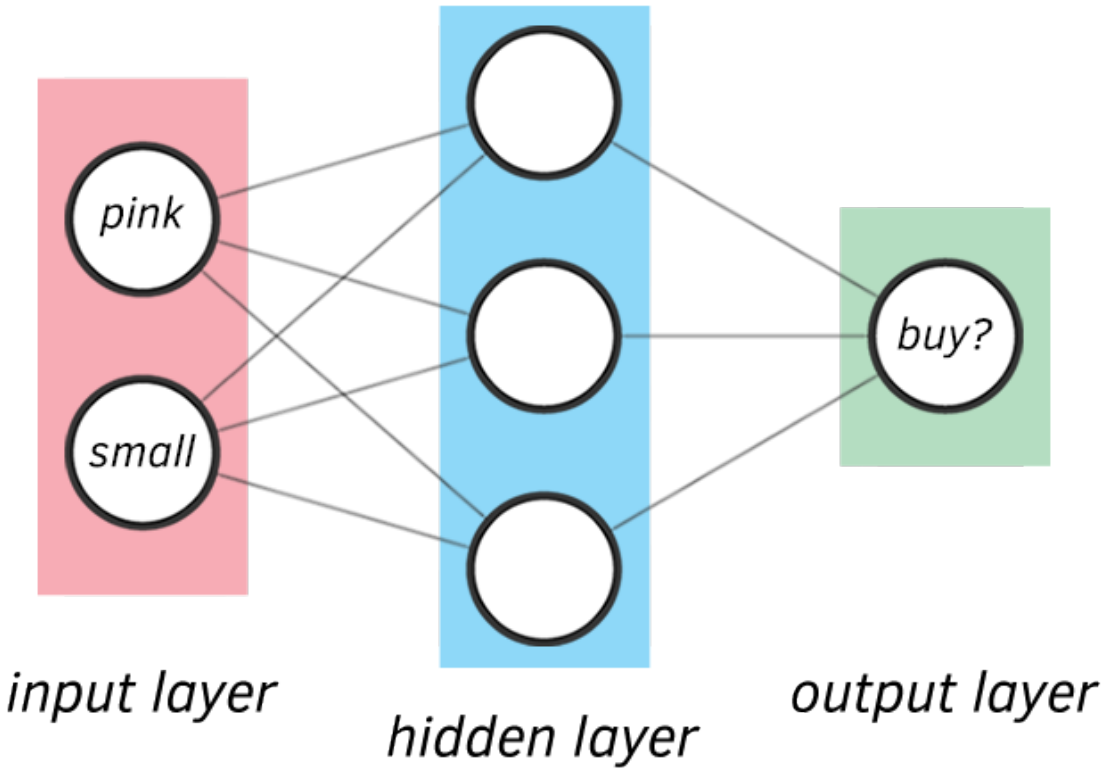
```
const X = tf.tensor2d([
  // pink, small
  [0.1, 0.1],
  [0.3, 0.3],
  [0.5, 0.6],
  [0.4, 0.8],
  [0.9, 0.1],
  [0.75, 0.4],
  [0.75, 0.9],
  [0.6, 0.9],
  [0.6, 0.75]
]);

// 0 - no buy, 1 - buy
const y = tf.tensor([0, 0, 1, 1, 0, 0, 1, 1, 1].map(y => oneHot(y, 2)));
```

Well done! You did well on incorporating your friend preferences.

[70] https://en.wikipedia.org/wiki/Normal_distribution
[71] https://js.tensorflow.org/api/latest/#Initializers

Building the model

Recall the Neural Network we're going to build:



Let's translate it into a TensorFlow.js model:

```
const model = tf.sequential();

model.add(
  tf.layers.dense({
    inputShape: [2],
    units: 3,
    activation: "relu"
  })
);

model.add(
  tf.layers.dense({
    units: 2,
    activation: "softmax"
  })
);
```

We have a *2-layer network* with an input layer containing *2* neurons, a hidden layer with *3* neurons and an output layer containing *2* neurons.

Note that we use ReLu activation function in the hidden layer and softmax for the output layer. We have 2 neurons in the output layer since we want to obtain how certain our Neural Network is in its buy/no-buy decision.

```
model.compile({
  optimizer: tf.train.adam(0.1),
  loss: "binaryCrossentropy",
  metrics: ["accuracy"]
});
```

We're using binary crossentropy[72] to measure the quality of the current weights/parameters of our model by measuring how "good" the predictions are.

Our training algorithm, Stochastic gradient descent[73], is trying to find weights that minimize the loss function. For our example, we're going to use the Adam optimizer[74].

Training

Now that our model is defined, we can use our training dataset to teach it about our friend preferences:

```
await model.fit(X, y, {
  shuffle: true,
  epochs: 20,
  callbacks: {
    onEpochEnd: async (epoch, logs) => {
      console.log("Epoch " + epoch);
      console.log("Loss: " + logs.loss + " accuracy: " + logs.acc);
    }
  }
});
```

We're shuffling the data before training and log the progress after each epoch is complete:

[72] https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/binary-crossentropy
[73] https://en.wikipedia.org/wiki/Stochastic_gradient_descent
[74] https://js.tensorflow.org/api/latest/#train.adam

```
Epoch 1
Loss: 0.703386664390564 accuracy: 0.5
Epoch 2
Loss: 0.6708164215087891 accuracy: 0.5555555820465088
Epoch 3
Loss: 0.6340110898017883 accuracy: 0.6666666865348816
Epoch 4
Loss: 0.6071969270706177 accuracy: 0.7777777910232544
...
Epoch 19
Loss: 0.08228953927755356 accuracy: 1
Epoch 20
Loss: 0.06922533363103867 accuracy: 1
```

After 20 epochs or so seems like the model has learned the preferences of your friend.

Evaluation

You save the model and send it over to your friend. After connecting to your friend computer, you find somewhat appropriate laptop and encode the information into the model:

```
const predProb = model.predict(tf.tensor2d([[0.1, 0.6]])).dataSync();
```

After waiting a few long milliseconds, you receive an answer:

```
0: 0.45
1: 0.55
```

The model agrees with you. It "thinks" that your friend should buy the laptop but it is not that certain about it. You did good!

Conclusion

Your friend seems happy with the results, and you're thinking of making millions with your model by selling it as a browser extension. Either way, you learned a lot about:

- The Perceptron model
- Why activation functions are needed and which one to use
- How to initialize the weights of your Neural Network models
- Build a simple Neural Network to solve a (somewhat) real problem

Run the complete source code for this tutorial right in your browser[75]

Laying back on the comfy pillow, you start thinking. Could I've used **Deep Learning** for this?

[75] https://codesandbox.io/s/simple-neural-network-with-tensorflow-js-7k0jr?fontsize=14

References

Reducing Loss: Gradient Descent[76]

Gradient descent and stochastic gradient descent from scratch[77]

Initializing neural networks[78]

Types of weight intializations[79]

What if do not use any activation function in the neural network?[80]

[76] https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent
[77] https://gluon.mxnet.io/chapter06_optimization/gd-sgd-scratch.html
[78] https://www.deeplearning.ai/ai-notes/initialization/
[79] https://www.deeplearningwizard.com/deep_learning/boosting_models_pytorch/weight_initialization_activation_functions/#types-of-weight-intializations
[80] https://stats.stackexchange.com/questions/267024/what-if-do-not-use-any-activation-function-in-the-neural-network

Customer churn prediction using Neural Networks

> TL;DR Learn about Deep Learning and create Deep Neural Network model to predict customer churn using TensorFlow.js. Learn how to preprocess string categorical data.

First day! You've landed this Data Scientist intern job at a large telecom company. You can't stop dreaming about the Lambos and designer clothes you're going to get once you're a Senior Data Scientist.

Even your mom is calling to remind you to put your Ph.D. in Statistics diploma on the wall. This is the life, who cares about that you're in your mid-30s and this is your first job ever.

Your team lead comes around, asking how do you enjoy the job and saying that he might have a task for you! You start imagining implementing complex statistical models from scratch, doing research, and adding cutting-edge methods but... Well, the reality is slightly different. He sent you a link to a CSV fail and asks you to predict customer churn. He suggests that you might to try to apply Deep Learning to the problem.

Your dream is starting now. Time do to some work!

Customer churn data

Our dataset *Telco Customer Churn* comes from Kaggle[81].

> "Predict behavior to retain customers. You can analyze all relevant customer data and develop focused customer retention programs." [IBM Sample Data Sets]

The data set includes information about:

- Customers who left within the last month – the column is called Churn
- Services that each customer has signed up for – phone, multiple lines, internet, online security, online backup, device protection, tech support, and streaming TV and movies
- Customer account information – how long they've been a customer, contract, payment method, paperless billing, monthly charges, and total charges
- Demographic info about customers – gender, age range, and if they have partners and dependents

[81] https://www.kaggle.com/blastchar/telco-customer-churn

It has 7,044 examples and 21 variables:

- **customerID**: Customer ID
- **gender**: Whether the customer is a male or a female
- **SeniorCitizen**: Whether the customer is a senior citizen or not (1, 0)
- **Partner**: Whether the customer has a partner or not (Yes, No)
- **Dependents**: Whether the customer has dependents or not (Yes, No)
- **tenure**: Number of months the customer has stayed with the company
- **PhoneService**: Whether the customer has a phone service or not (Yes, No)
- **MultipleLines**: Whether the customer has multiple lines or not (Yes, No, No phone service)
- **InternetService**: Customer's internet service provider (DSL, Fiber optic, No)
- **OnlineSecurity**: Whether the customer has online security or not (Yes, No, No internet service)
- **OnlineBackup**: Whether the customer has online backup or not (Yes, No, No internet service)
- **DeviceProtection**: Whether the customer has device protection or not (Yes, No, No internet service)
- **TechSupport**: Whether the customer has tech support or not (Yes, No, No internet service)
- **StreamingTV**: Whether the customer has streaming TV or not (Yes, No, No internet service)
- **StreamingMovies**: Whether the customer has streaming movies or not (Yes, No, No internet service)
- **Contract**: The contract term of the customer (Month-to-month, One year, Two year)
- **PaperlessBilling**: Whether the customer has paperless billing or not (Yes, No)
- **PaymentMethod**: The customer's payment method (Electronic check, Mailed check, Bank transfer (automatic), Credit card (automatic))
- **MonthlyCharges**: The amount charged to the customer monthly
- **TotalCharges**: The total amount charged to the customer
- **Churn**: Whether the customer churned or not (Yes or No)

We'll use Papa Parse[82] to load the data:

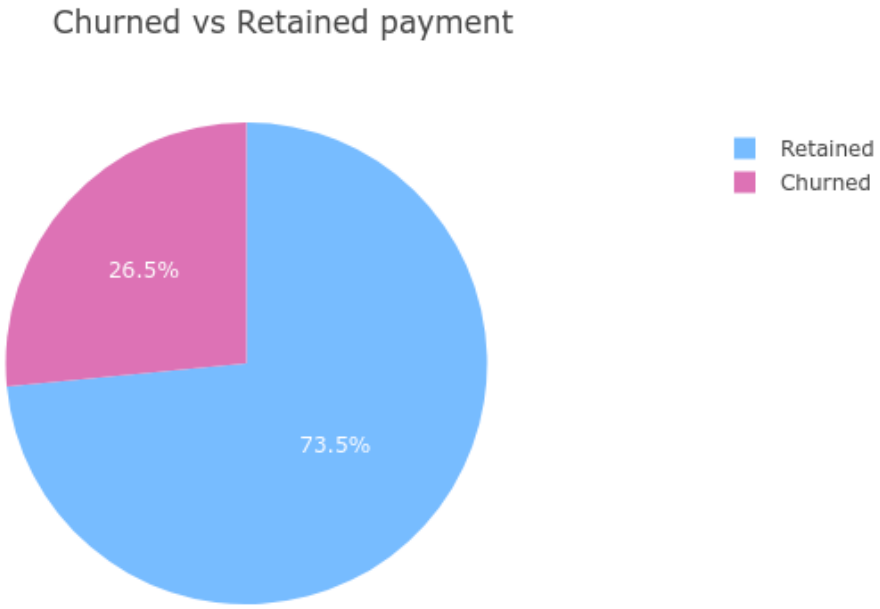
```
const prepareData = async () => {
  const csv = await Papa.parsePromise(
    "https://raw.githubusercontent.com/curiousily/Customer-Churn-Detection-with-Tens\
orFlow-js/master/src/data/customer-churn.csv"
  );

  const data = csv.data;
  return data.slice(0, data.length - 1);
};
```

Note that we ignore the last row since it is empty.

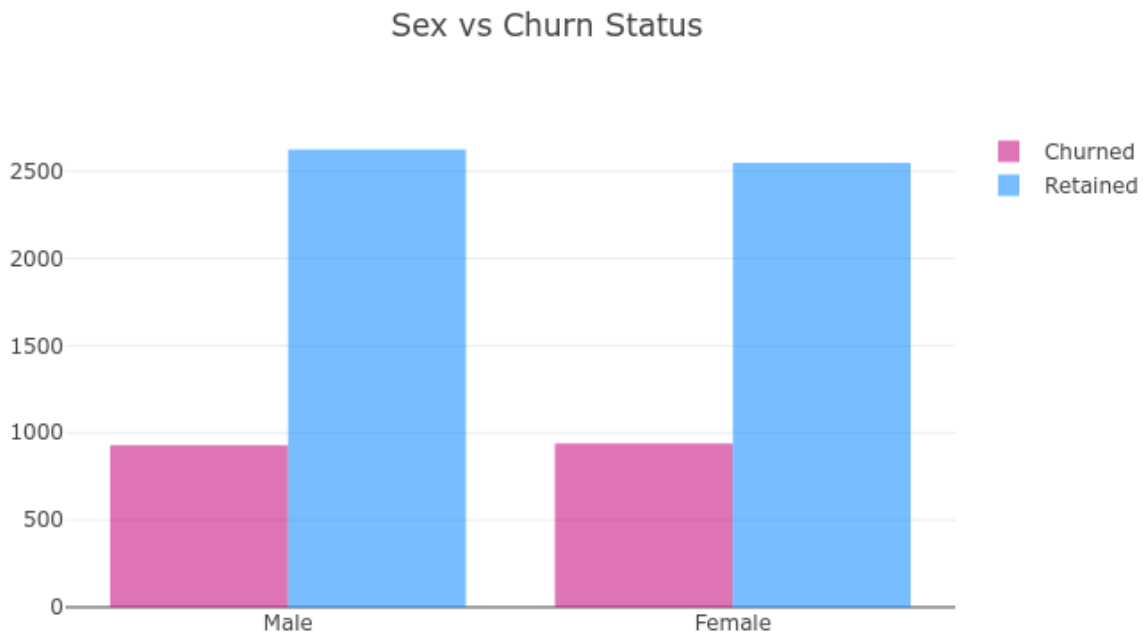
[82] https://www.papaparse.com/

Exploration

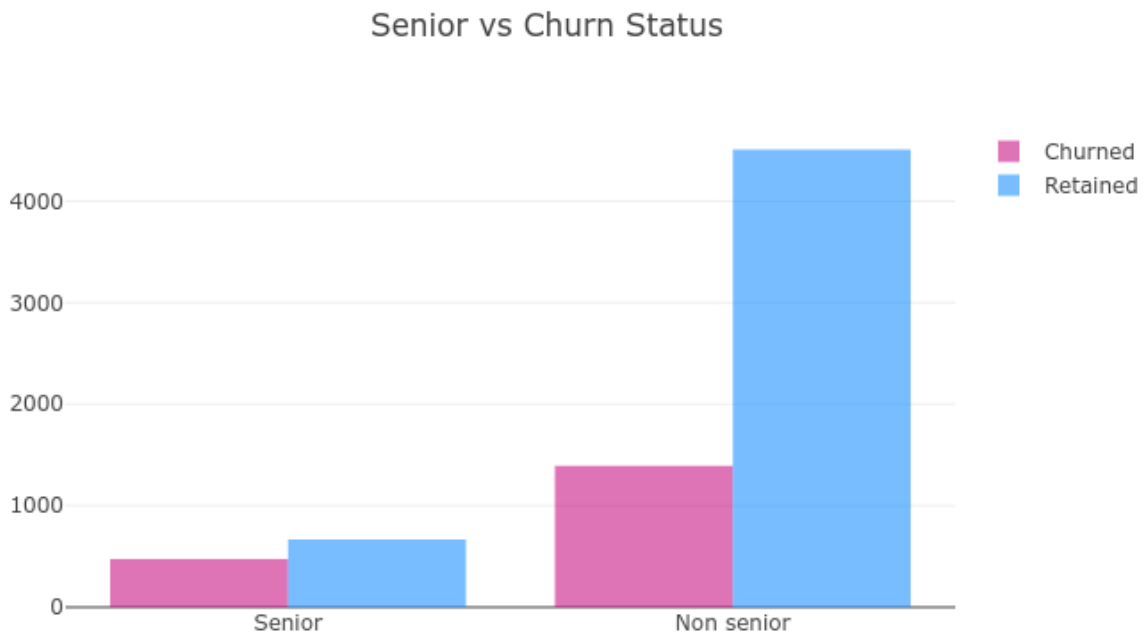
Let's get a feeling of our dataset. How many of the customers churned?



About 74% of the customers are still using the company services. We have a very unbalanced dataset. Does gender play a role in losing customers?

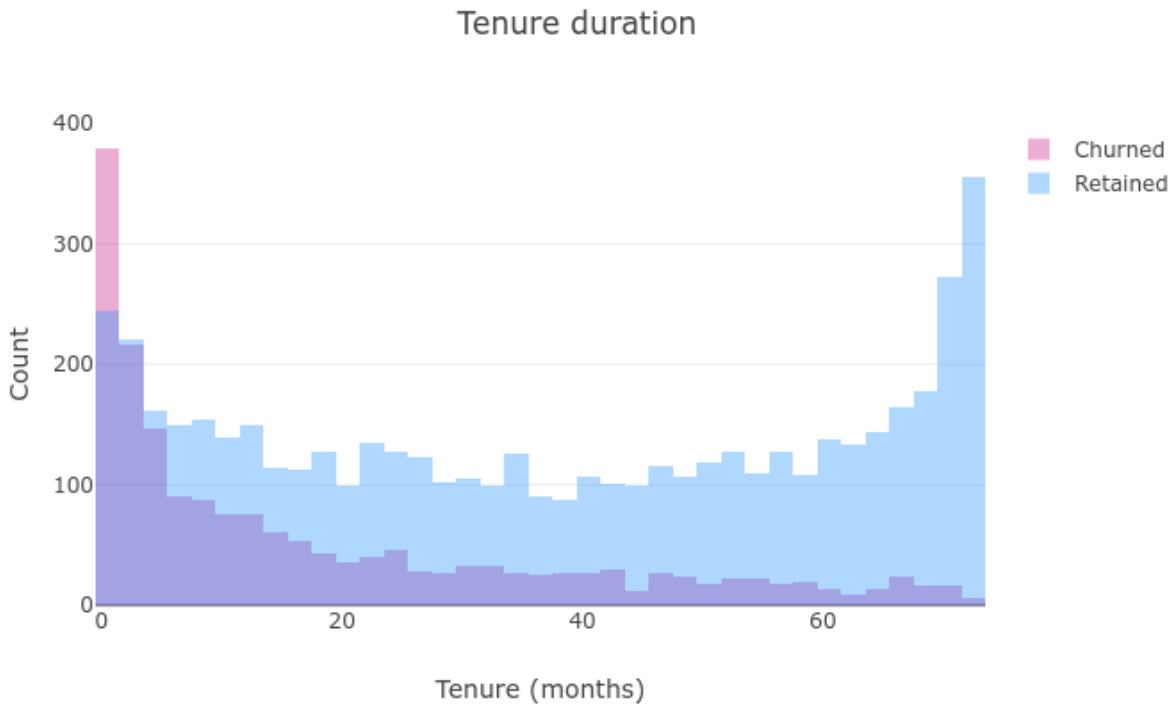


Seems like it doesn't. We have about the same amount of female and male customers. How about seniority?



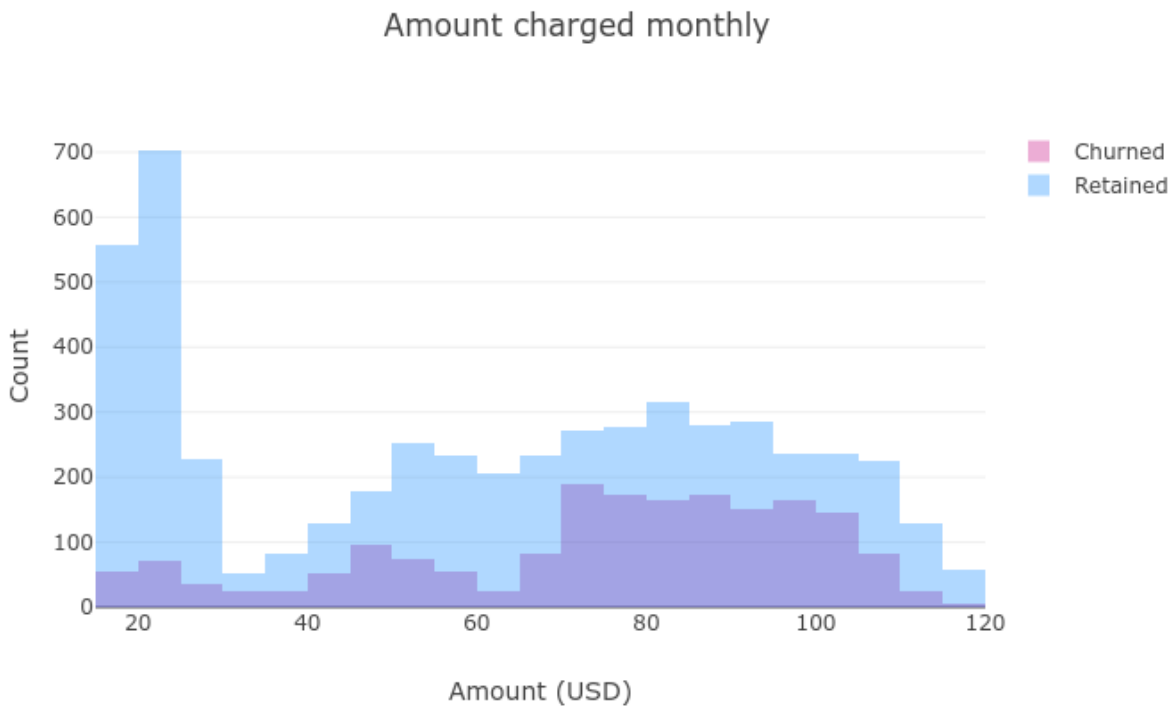
About 20% of the customers are senior, and they are much more likely to churn, compared to the nonseniors.

For how long customers stay with the company?



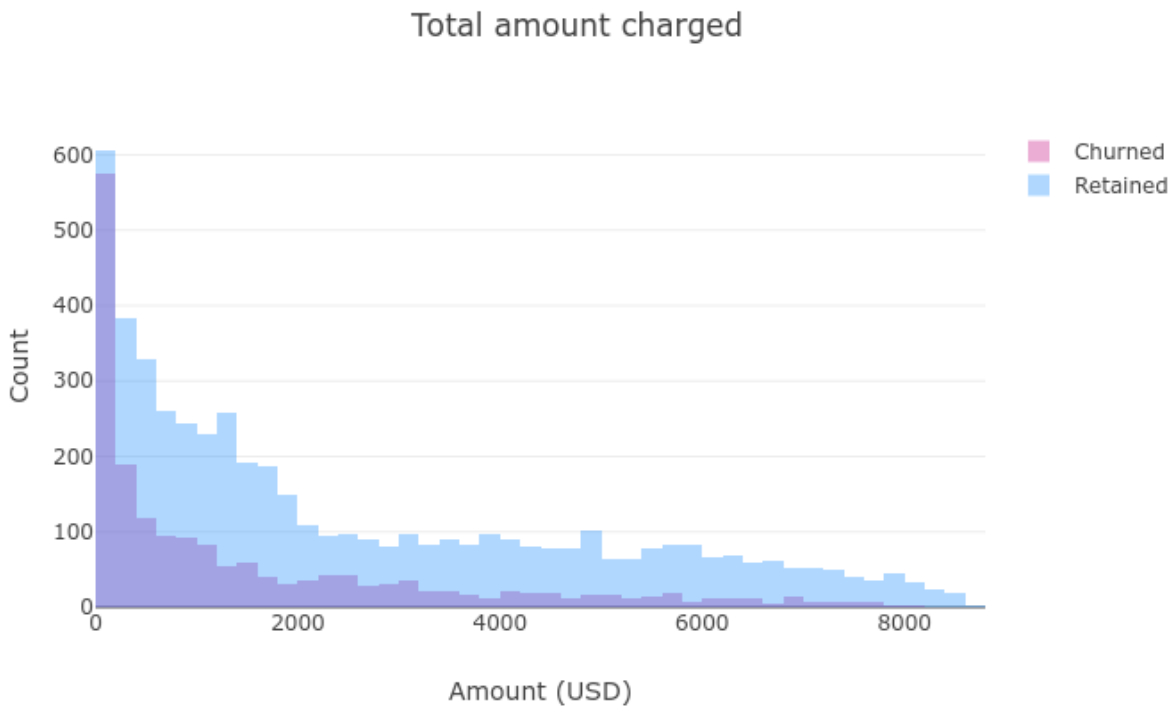
Seems like the more you stay, the more likely you're to stay with the company.

How do monthly charges affect churn?



A customer with low monthly charges (< $30) produce is much more likely to be retained.

How about the total amount charged per customer?



The higher the total amount charged by the company, the more likely it is for this customer to be retained.

Our dataset has a total of 21 features, and we didn't look through all of those. However, we found some interesting stuff.

We've learned that SeniorCitizen, tenure, MonthlyCharges, and TotalCharges are somewhat correlated with the churn status. We'll use them for our model!

Deep Learning

Deep Learning is a subset of Machine Learning, using Deep Artificial Neural Networks[83] as a primary model to solve a variety of tasks.

To obtain a Deep Neural Network, take a Neural Network with one hidden layer (shallow Neural Network) and add more layers. That's the definition of a Deep Neural Network - Neural Network with more than one hidden layer!

In Deep Neural Networks, each layer of neurons is trained on the features/outputs of the previous layer. Thus, you can create a feature hierarchy of increasing abstraction and learn complex concepts.

[83] https://en.wikipedia.org/wiki/Deep_learning#Deep_neural_networks

These networks are very good at discovering patterns within raw data (images, texts, video, and audio recordings), which is the most amounts of data we have. For example, Deep Learning can take millions of images and categorize them into photos of your grandma, funny cats, and delicious cakes.

Deep Neural Nets are holding state-of-the-art scores[84] on a variety of important problems. Examples are image recognition, image segmentation, sound recognition, recommender systems, natural language processing, etc.

So basically, Deep Learning is Large Neural Networks. Why now? Why Deep Learning wasn't practical before?

- *Most real-world applications of Deep Learning require large amounts of labeled data*: developing a driverless car might require thousands of hours of video.
- *Training models with large amounts of parameters (weights) requires substantial computing power*: special purpose hardware in the form of GPUs and TPUs offers massively parallel computations, suitable for Deep Learning.
- *Big companies have been storing your data for a while now*: they want to monetize it.
- *We learned (kinda) how to initialize the weights of the neurons in the Neural Network models*: mostly using small random values
- *We have better regularization techniques* (e.g. Dropout[85])

Last but not least, we have software that is performant and (sometimes) easy to use. Libraries like TensorFlow[86], PyTorch[87], MXNet[88] and Chainer[89] allows practitioners to develop, analyze, test and deploy models of varying complexity and reuse work done by other practitioners and researchers.

Predicting customer churn

Let's use the "all-powerful" Deep Learning machinery to predict which customers are going to churn. First, we need to do some data preprocessing since a lot of the features are categorical.

Data preprocessing

We'll use all numerical (except *customerID*) and the following categorical features:

[84] https://paperswithcode.com/sota
[85] https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf
[86] https://www.tensorflow.org/
[87] https://pytorch.org/
[88] https://mxnet.apache.org/
[89] https://chainer.org/

```
const categoricalFeatures = new Set([
  "TechSupport",
  "Contract",
  "PaymentMethod",
  "gender",
  "Partner",
  "InternetService",
  "Dependents",
  "PhoneService",
  "TechSupport",
  "StreamingTV",
  "PaperlessBilling"
]);
```

Let's create training and testing datasets from our data:

```
const [xTrain, xTest, yTrain, yTest] = toTensors(
  data,
  categoricalFeatures,
  0.1
);
```

Here's how we create our Tensors:

```
const toTensors = (data, categoricalFeatures, testSize) => {
  const categoricalData = {};
  categoricalFeatures.forEach(f => {
    categoricalData[f] = toCategorical(data, f);
  });

  const features = [
    "SeniorCitizen",
    "tenure",
    "MonthlyCharges",
    "TotalCharges"
  ].concat(Array.from(categoricalFeatures));

  const X = data.map((r, i) =>
    features.flatMap(f => {
      if (categoricalFeatures.has(f)) {
        return categoricalData[f][i];
      }
```

```
      return r[f];
    })
  );

  const X_t = normalize(tf.tensor2d(X));

  const y = tf.tensor(toCategorical(data, "Churn"));

  const splitIdx = parseInt((1 - testSize) * data.length, 10);

  const [xTrain, xTest] = tf.split(X_t, [splitIdx, data.length - splitIdx]);
  const [yTrain, yTest] = tf.split(y, [splitIdx, data.length - splitIdx]);

  return [xTrain, xTest, yTrain, yTest];
};
```

First, we use the function `toCategorical()` to convert categorical features into one-hot[90] encoded vectors. We do that by converting the string values into numbers and use tf.oneHot()[91] to create the vectors.

We create a 2-dimensional Tensor from our features (categorical and numerical) and normalize it[92]. Another, one-hot encoded, Tensor is made from the Churn column.

Finally, we split the data into training and testing datasets and return the results. How do we encode categorical variables?

```
const toCategorical = (data, column) => {
  const values = data.map(r => r[column]);
  const uniqueValues = new Set(values);

  const mapping = {};

  Array.from(uniqueValues).forEach((i, v) => {
    mapping[i] = v;
  });

  const encoded = values
    .map(v => {
      if (!v) {
        return 0;
```

[90] https://en.wikipedia.org/wiki/One-hot
[91] https://js.tensorflow.org/api/latest/#oneHot
[92] https://en.wikipedia.org/wiki/Feature_scaling

```
      }
      return mapping[v];
    })
    .map(v => oneHot(v, uniqueValues.size));

  return encoded;
};
```

First, we extract a vector of all values for the feature. Next, we obtain the unique values and create a string to int mapping from it.

Note that we check for missing values and encode those as 0. Finally, we one-hot encode each value.

Here are the remaining utility functions:

```
// normalized = (value - min_value) / (max_value - min_value)
const normalize = tensor =>
  tf.div(
    tf.sub(tensor, tf.min(tensor)),
    tf.sub(tf.max(tensor), tf.min(tensor))
  );

const oneHot = (val, categoryCount) =>
  Array.from(tf.oneHot(val, categoryCount).dataSync());
```

Building a Deep Neural Network

We'll wrap the building and training of our model into a function called `trainModel()`:

```
const trainModel = async (xTrain, yTrain) => {
  ...
}
```

Let's create a Deep Neural Network using the sequential model API[93] in TensorFlow:

[93] https://js.tensorflow.org/api/latest/#sequential

```
const model = tf.sequential();
model.add(
  tf.layers.dense({
    units: 32,
    activation: "relu",
    inputShape: [xTrain.shape[1]]
  })
);

model.add(
  tf.layers.dense({
    units: 64,
    activation: "relu"
  })
);

model.add(tf.layers.dense({ units: 2, activation: "softmax" }));
```

Our Deep Neural Network has two hidden layers with 32 and 64 neurons, respectively. Each layer has a ReLU[94] activation function.

Time to compile[95] our model:

```
model.compile({
  optimizer: tf.train.adam(0.001),
  loss: "binaryCrossentropy",
  metrics: ["accuracy"]
});
```

We'll train our model using the Adam optimizer[96] and measure our error using Binary Crossentropy[97].

Training

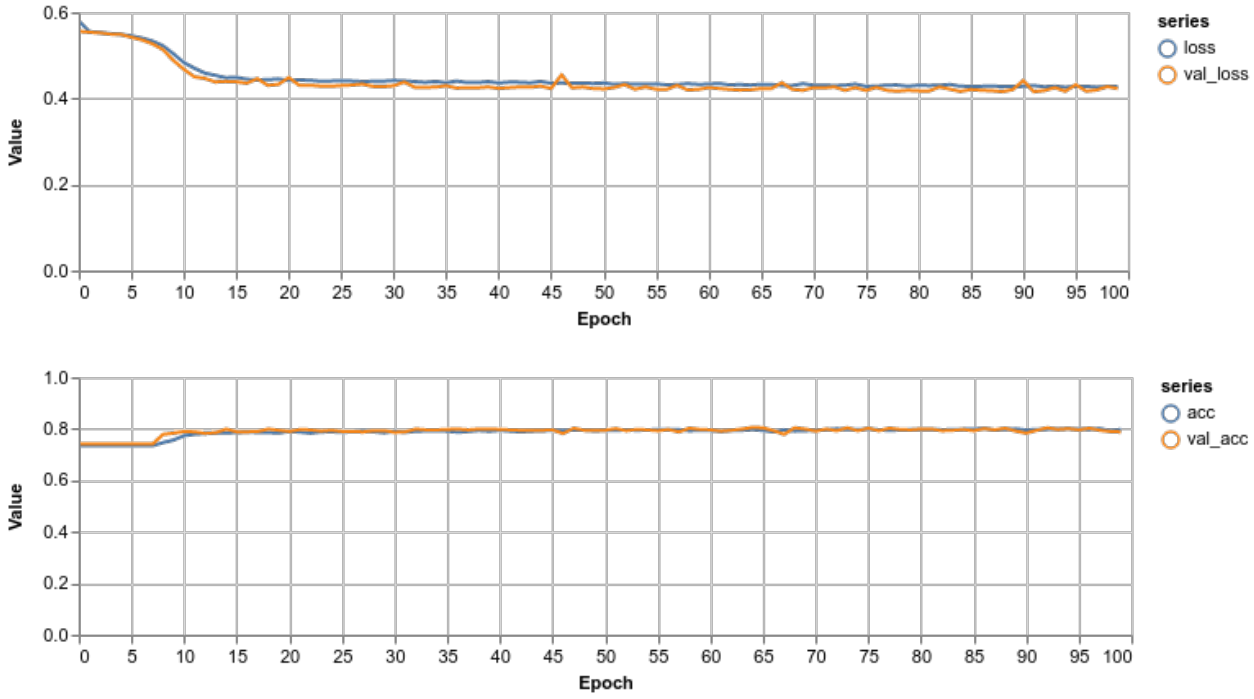
Finally, we'll pass the training data to the fit[98] method of our model and train for *100* epochs, *shuffle the data*, and use *10%* of it for validation. We'll visualize the training progress using tfjs-vis[99]:

[94] https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
[95] https://js.tensorflow.org/api/latest/#tf.LayersModel.compile
[96] https://js.tensorflow.org/api/latest/#train.adam
[97] https://js.tensorflow.org/api/latest/#metrics.binaryCrossentropy
[98] https://js.tensorflow.org/api/latest/#tf.Sequential.fit
[99] https://github.com/tensorflow/tfjs-vis

```
const lossContainer = document.getElementById("loss-cont");

await model.fit(xTrain, yTrain, {
  batchSize: 32,
  epochs: 100,
  shuffle: true,
  validationSplit: 0.1,
  callbacks: tfvis.show.fitCallbacks(
    lossContainer,
    ["loss", "val_loss", "acc", "val_acc"],
    {
      callbacks: ["onEpochEnd"]
    }
  )
});
```

Let's train our model:

```
const model = await trainModel(xTrain, yTrain);
```



It seems like our model is learning during the first ten epochs and plateaus after that.

Evaluation

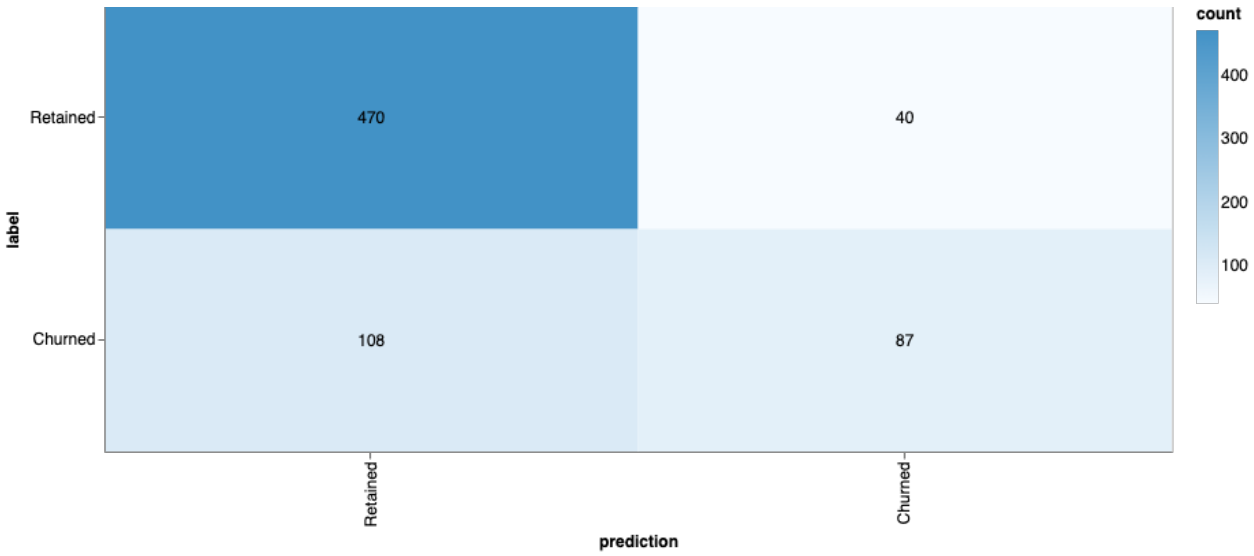
Let's evaluate our model on the test data:

```
const result = model.evaluate(xTest, yTest, {
  batchSize: 32
});

// loss
result[0].print();

// accuracy
result[1].print();
```

```
Tensor
    0.44808024168014526
Tensor
    0.7929078340530396
```

The model has an accuracy of 79.2% on the test data. Let's have a look at what kind of mistakes it makes using the confusion matrix:

```
const preds = model.predict(xTest).argMax(-1);
const labels = yTest.argMax(-1);
const confusionMatrix = await tfvis.metrics.confusionMatrix(labels, preds);
const container = document.getElementById("confusion-matrix");
tfvis.render.confusionMatrix(container, {
  values: confusionMatrix,
  tickLabels: ["Retained", "Churned"]
});
```



It seems like our model is overconfident in predicting retained customers. Depending on your needs, you might try to tune the model, and predict retained customers better.

Conclusion

Great job! You just built a Deep Neural Network that predicts customer churn with ~80% accuracy. Here's what you've learned:

- What is Deep Learning
- What is the difference between shallow Neural Networks and Deep Neural Networks
- Preprocess string categorical data
- Build and evaluate a Deep Neural Network in TensorFlow.js

But can it be that Deep Learning is even more powerful? So powerful that it can understand images?

References

- Deep Learning & Artificial Neural Networks[100]
- What Is Deep Learning?[101]
- A Beginner's Guide to Neural Networks and Deep Learning[102]

[100] https://machinelearningmastery.com/what-is-deep-learning/
[101] https://www.mathworks.com/discovery/deep-learning.html
[102] https://skymind.ai/wiki/neural-network

Alien vs Predator image classification using Deep Convolutional Neural Networks

> TL;DR Learn how to create a Deep Convolutional Neural Network in TensorFlow.js and use it to classify images of Aliens and Predators

Staying alone in the corner, you and your motion detector. Total silence. Then you hear an unwanted beep and see a dot. 100 feet. 90 feet. You prepare yourself. This place is very dark, you think. You see something in the distance. Another beep - 20 feet. 10 feet. The thing is much more recognizable now, but what is it?

Here's what you'll learn:

- Load images from remote URLs in the browser
- Transform (turn to greyscale, resize and rescale) image pixel data into Tensors
- Build a simple Deep Convolutional Neural Network from scratch using TensorFlow.js
- Visualize filters (what the network learns) of convolutional layers

Let's build a model that can distinguish between an Alien and a Predator!

Run the complete source code for this tutorial right in your browser[103]

Source code on GitHub[104]

Data

Our dataset *Alien vs. Predator images* comes from Kaggle[105]. Each class (alien and predator) contains 247 training images and 100 validation images. The images are JPG files with a size of around 250x250 pixels. Let's use those to build a Neural Network that can classify them!

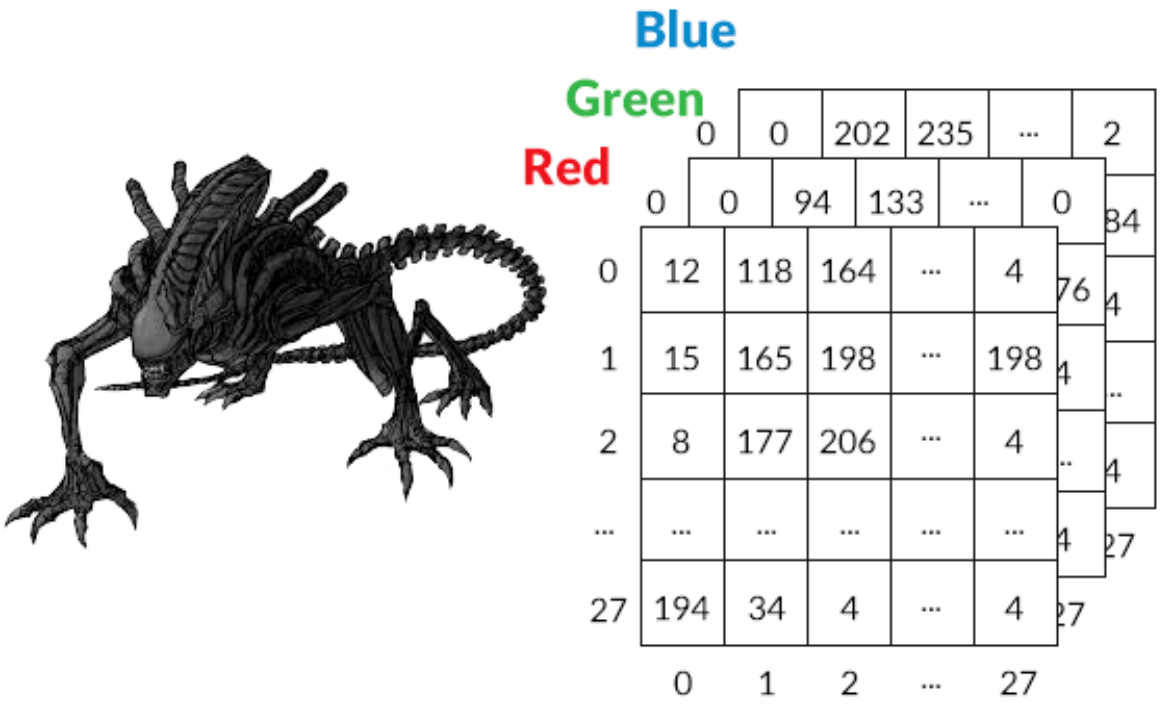
[103] https://codesandbox.io/s/alien-vs-predator-tensorflow-js-classification-ouvt8?fontsize=14
[104] https://github.com/curiousily/alien-vs-predator-tensorflow-js-classification
[105] https://www.kaggle.com/pmigdal/alien-vs-predator-images

Convolutional Neural Networks

Building models that understand the real-world from raw image data has been a dream for many computer scientists. In recent years, this dream has been somewhat achieved.

Using traditional Machine Learning models for computer vision requires a manual definition of features. Convolutional Neural Networks allow for automation of this step. Those features are learned during the training from the raw image data.

Computers store images in width x height x 3 (or 4) tensors. The last dimension represents the colors (Red Green Blue) and a possible alpha channel. This poses a challenge for regular Neural Networks - every pixel (and color data) will be stored as a neuron. Some of the "getting started" image datasets are ok. CIFAR-10 has images of 32x32x3 = 3072 neurons. But we all know that images in the wild are huge!



Convolutional Neural Networks (CNNs/ConvNets) take advantage of the two-dimensional image data. Still, they can be used with other types of data[106]. CNNs consists of **Convolutional**, **Pooling** and **Fully-Connected** (regular) layers.

Convolutional layers have neurons arranged in 3 dimensions: width, height, depth. The parameters (those who we try to learn during training) consist of a set of learnable filters. Every filter covers a small area (width x height) but extends through the full depth.

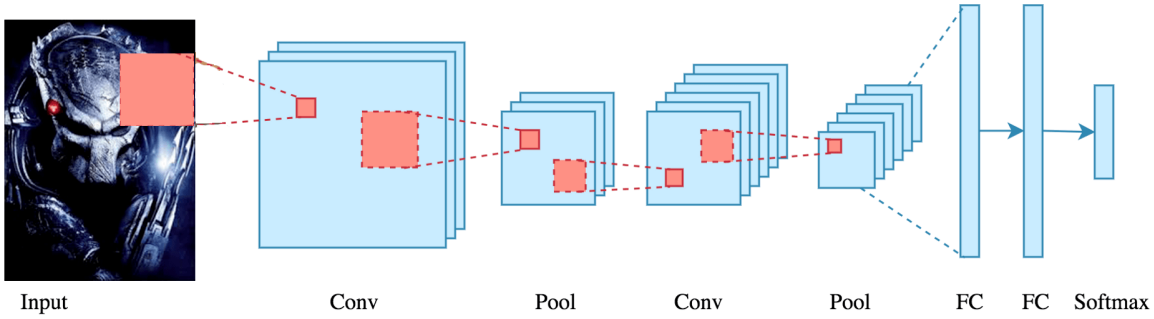
During training, we slide each filter across the image's height and weight and compute dot products

[106] https://arxiv.org/pdf/1606.01781.pdf

between the filter and input.

One common way to design a Convolutional Neural Network for classification or regression is:

- Input Layer - holds the raw pixel values of the image
- Conv Layer - computes the dot product between filters and input data
- Pool Layer - downsamples the width and height of the input
- Fully-Connected Layer - delivers the predictions, based on the task at hand (classification or regression)



Classifying Aliens and Predators

We'll load the image data from GitHub[107], do some preprocessing and train a CNN model with it.

Let's look at an example image of an Alien:

[107] https://github.com/curiousily/tfjs-data

And a Predator:

One thing to note from these examples is that the type of images can be quite different. Let's see if the model we're going to build can handle that.

Data preprocessing

We'll use the image-pixels[108] package to load pixel data for our images:

```
const loadImages = async urls => {
  const imageData = await pixels.all(urls, { clip: [0, 0, 192, 192] });
  return imageData.map(d =>
    rescaleImage(resizeImage(toGreyScale(tf.browser.fromPixels(d))))
  );
};
```

We use the clip option to turn each image into 192x192 square. We convert each image into a Tensor using tf.browser.fromPixels[109], turn into greyscale, resize it and rescale:

```
const toGreyScale = image => image.mean(2).expandDims(2);
```

To turn an image with color into greyscale we just take the mean from the values in the color channels.

[108] https://www.npmjs.com/package/image-pixels
[109] https://js.tensorflow.org/api/1.2.7/#browser.fromPixels

```
const TENSOR_IMAGE_SIZE = 28;

const resizeImage = image =>
  tf.image.resizeBilinear(image, [TENSOR_IMAGE_SIZE, TENSOR_IMAGE_SIZE]);
```

The resizing is done by tf.image.resizeBilinear[110] and converts the image to a 28x28x3 Tensor.

```
const rescaleImage = image => {
  const expandedImage = image.expandDims(0);

  return expandedImage
    .toFloat()
    .div(tf.scalar(127))
    .sub(tf.scalar(1));
};
```

Each pixel across each color channel has a value in the range of 0-255. We convert that to a range of -1 to 1 and add a dimension to our image Tensor.

Let's load the images:

```
const trainAlienImages = await loadImages(trainAlienImagePaths);
const trainPredatorImages = await loadImages(trainPredatorImagePaths);

const testAlienImages = await loadImages(validAlienImagePaths);
const testPredatorImages = await loadImages(validPredatorImagePaths);
```

And prepare the training and test data (features and labels):

```
const trainAlienTensors = tf.concat(trainAlienImages);

const trainPredatorTensors = tf.concat(trainPredatorImages);

const trainAlienLabels = tf.tensor1d(
  _.times(TRAIN_IMAGES_PER_CLASS, _.constant(0)),
  "int32"
);

const trainPredatorLabels = tf.tensor1d(
  _.times(TRAIN_IMAGES_PER_CLASS, _.constant(1)),
  "int32"
```

[110] https://js.tensorflow.org/api/1.2.7/#image.resizeBilinear

```
);

const testAlienTensors = tf.concat(testAlienImages);

const testPredatorTensors = tf.concat(testPredatorImages);

const testAlienLabels = tf.tensor1d(
  _.times(VALID_IMAGES_PER_CLASS, _.constant(0)),
  "int32"
);

const testPredatorLabels = tf.tensor1d(
  _.times(VALID_IMAGES_PER_CLASS, _.constant(1)),
  "int32"
);

const xTrain = tf.concat([trainAlienTensors, trainPredatorTensors]);

const yTrain = tf.concat([trainAlienLabels, trainPredatorLabels]);

const xTest = tf.concat([testAlienTensors, testPredatorTensors]);

const yTest = tf.concat([testAlienLabels, testPredatorLabels]);
```

Note that we preserve the order of the training data, so we need to shuffle it later.

Building a Deep Convolutional Neural Network

Let's have a look at how to build our CNN model:

```
const { conv2d, maxPooling2d, flatten, dense } = tf.layers;

const model = tf.sequential();

model.add(
  conv2d({
    name: "first-conv-layer",
    inputShape: [TENSOR_IMAGE_SIZE, TENSOR_IMAGE_SIZE, 1],
    kernelSize: 5,
    filters: 8,
    strides: 1,
    activation: "relu",
```

```
    kernelInitializer: "varianceScaling"
  })
);
model.add(maxPooling2d({ poolSize: 2, strides: 2 }));

model.add(
  conv2d({
    name: "second-conv-layer",
    kernelSize: 5,
    filters: 16,
    strides: 1,
    activation: "relu",
    kernelInitializer: "varianceScaling"
  })
);
model.add(maxPooling2d({ poolSize: 2, strides: 2 }));

model.add(flatten());

model.add(
  dense({
    units: 1,
    kernelInitializer: "varianceScaling",
    activation: "sigmoid"
  })
);

model.compile({
  optimizer: tf.train.adam(0.0001),
  loss: "binaryCrossentropy",
  metrics: ["accuracy"]
});
```

We're stacking two pairs of convolutional[111] and max pooling[112] layers. Each convolutional layer applies to 5x5 patches (*kernelSize*). Our downsizing operation reduces the size of the input by 2 (poolSize) and moves the application window by 2 pixels (right and down).

After computing all convolutions, we're flattening the input (converting it to a single dimension) using tf.layers.flatten()[113]. The output layer has a single neuron and sigmoid[114] activation function.

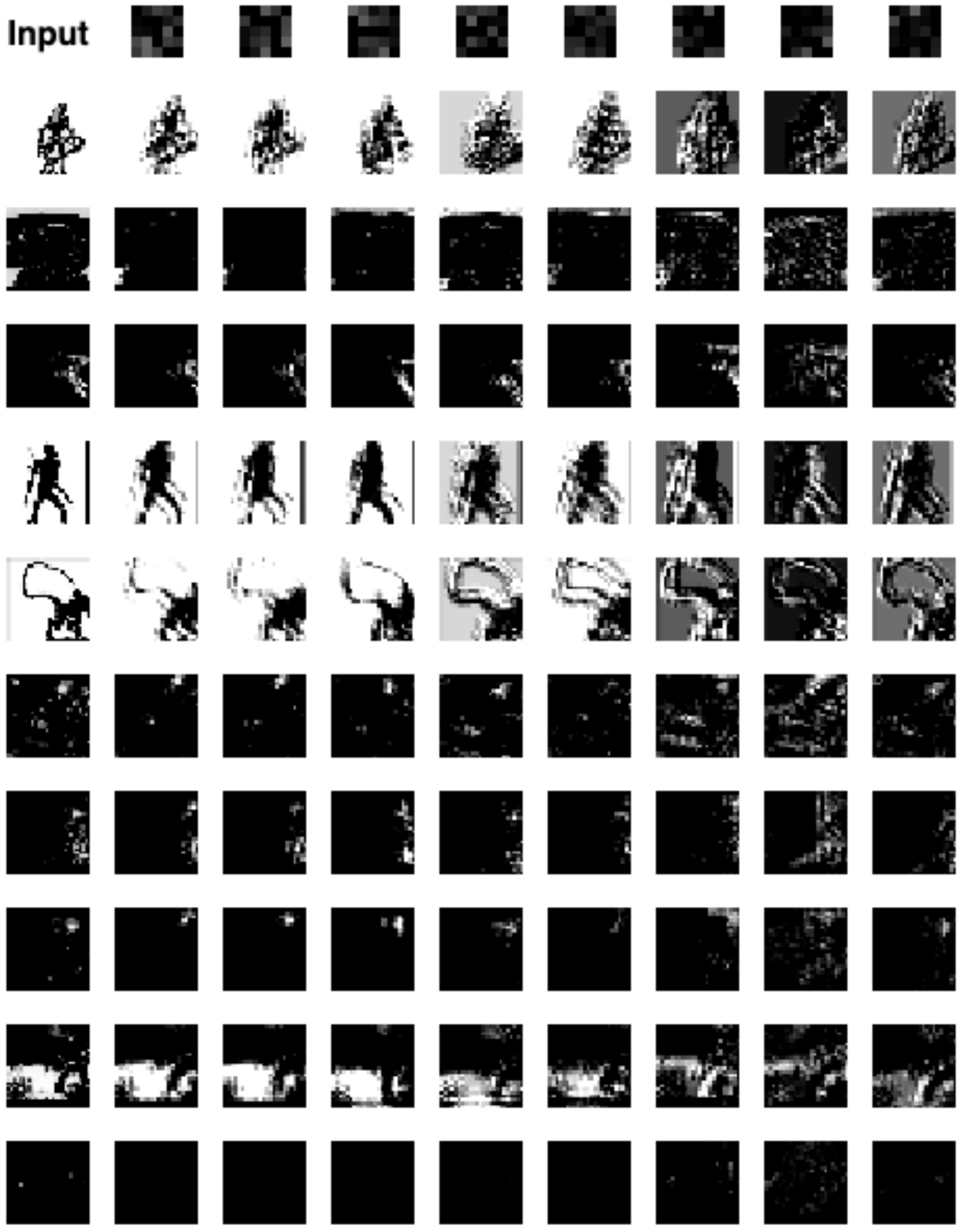
[111] https://js.tensorflow.org/api/1.2.7/#layers.conv2d
[112] https://js.tensorflow.org/api/1.2.7/#layers.maxPooling2d
[113] https://js.tensorflow.org/api/latest/#layers.flatten
[114] https://en.wikipedia.org/wiki/Sigmoid_function

We use Adam optimizer[115] and binary cross-entropy[116] loss function.

All neurons are initialized (*kernelInitializer*) using varianceScaling[117].

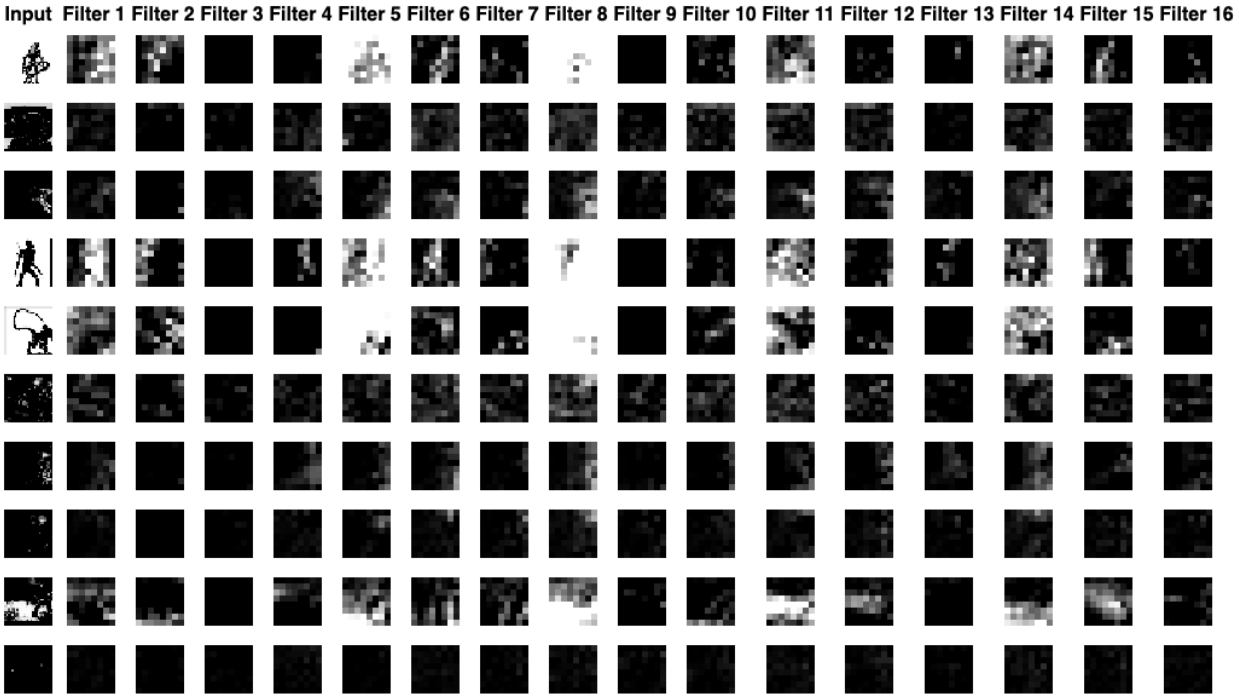
Let's have a look at the filters at the first convolutional layer before our model has seen the data (using the first 5 Alien and Predator images from the test set):

[115] https://js.tensorflow.org/api/latest/#train.adam
[116] https://js.tensorflow.org/api/latest/#metrics.binaryCrossentropy
[117] https://js.tensorflow.org/api/latest/#initializers.varianceScaling



Looks like nothing is looking important to the untrained first layer. Let's have a look at the second

one:



The same thing, the only difference is the size of the patches is much smaller. Let's train our model and have another look at those filters.

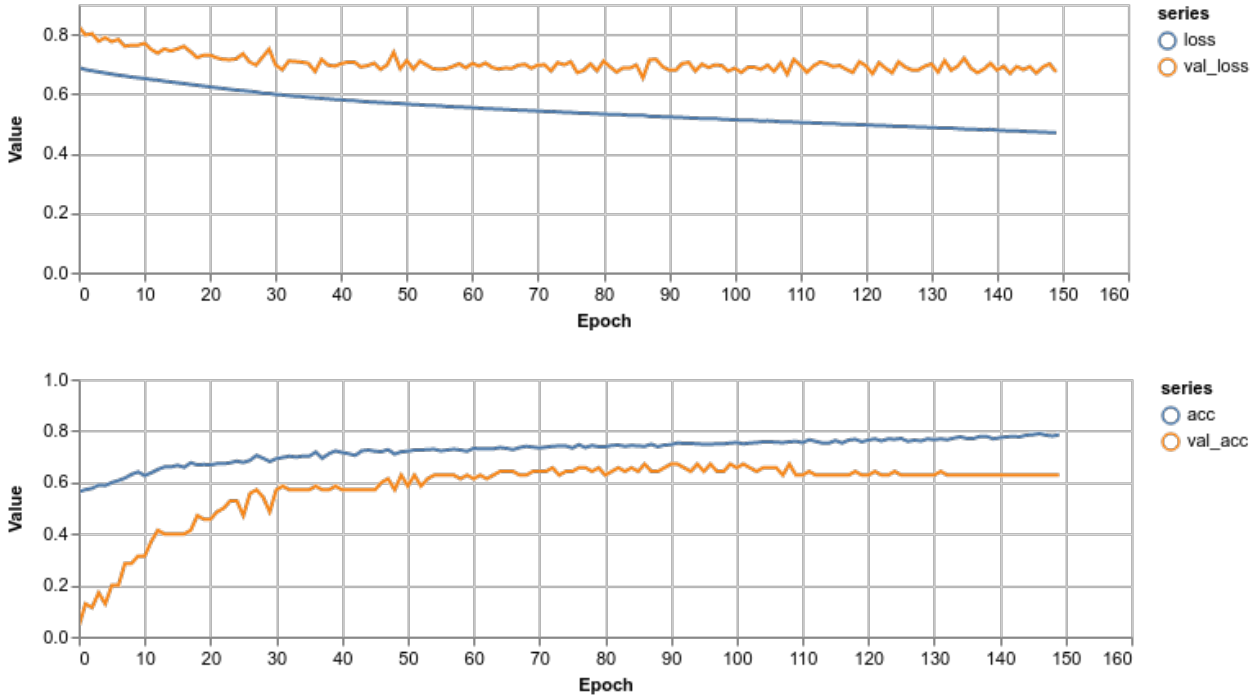
Training

Training a CNN is not any different from a regular Deep Neural Network using TensorFlow.js. Let's get to it:

```
const lossContainer = document.getElementById("loss-cont");

await model.fit(xTrain, yTrain, {
  batchSize: 32,
  validationSplit: 0.1,
  shuffle: true,
  epochs: 150,
  callbacks: tfvis.show.fitCallbacks(
    lossContainer,
    ["loss", "val_loss", "acc", "val_acc"],
    {
      callbacks: ["onEpochEnd"]
    }
  )
});
```

Note that we shuffle the data, before training and using 10% of it for validation.

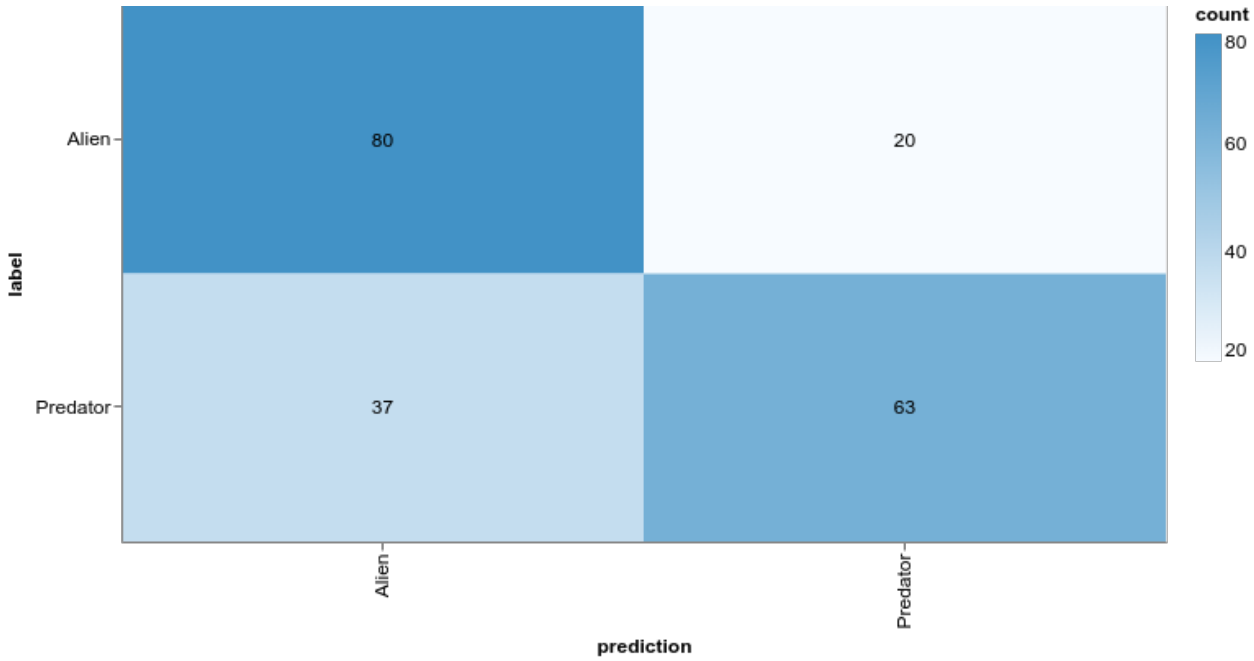
We have two perfectly balanced (equal number of examples) classes of images. We would expect that a dummy (random) model would predict with an accuracy of about 50%. How did we do?



Our model plateaus at around 62% accuracy which is not great, but we don't have a particularly large dataset. Let's dive a bit deeper into the performance.

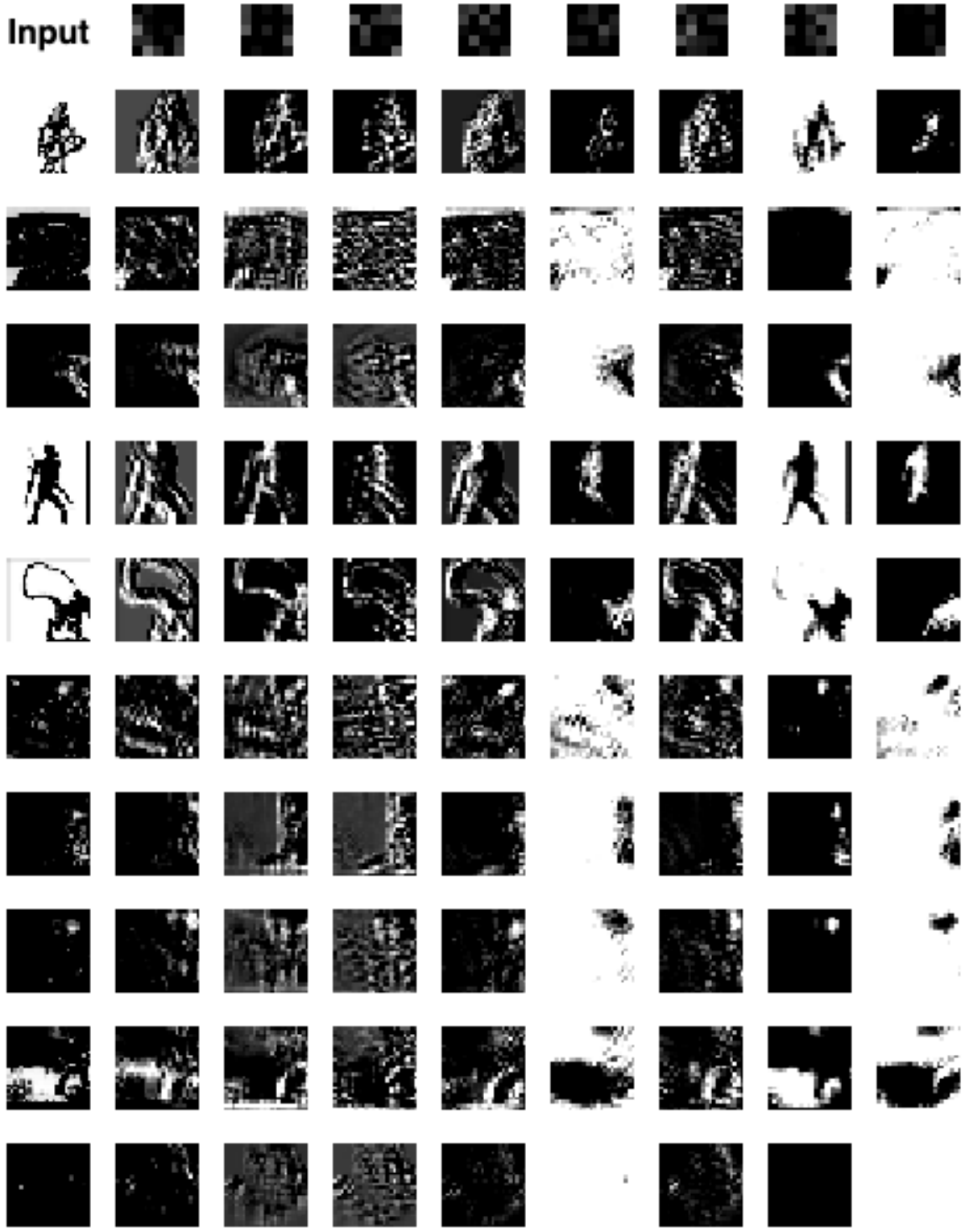
Evaluation

Let's see how well we predict each class using the confusion matrix:



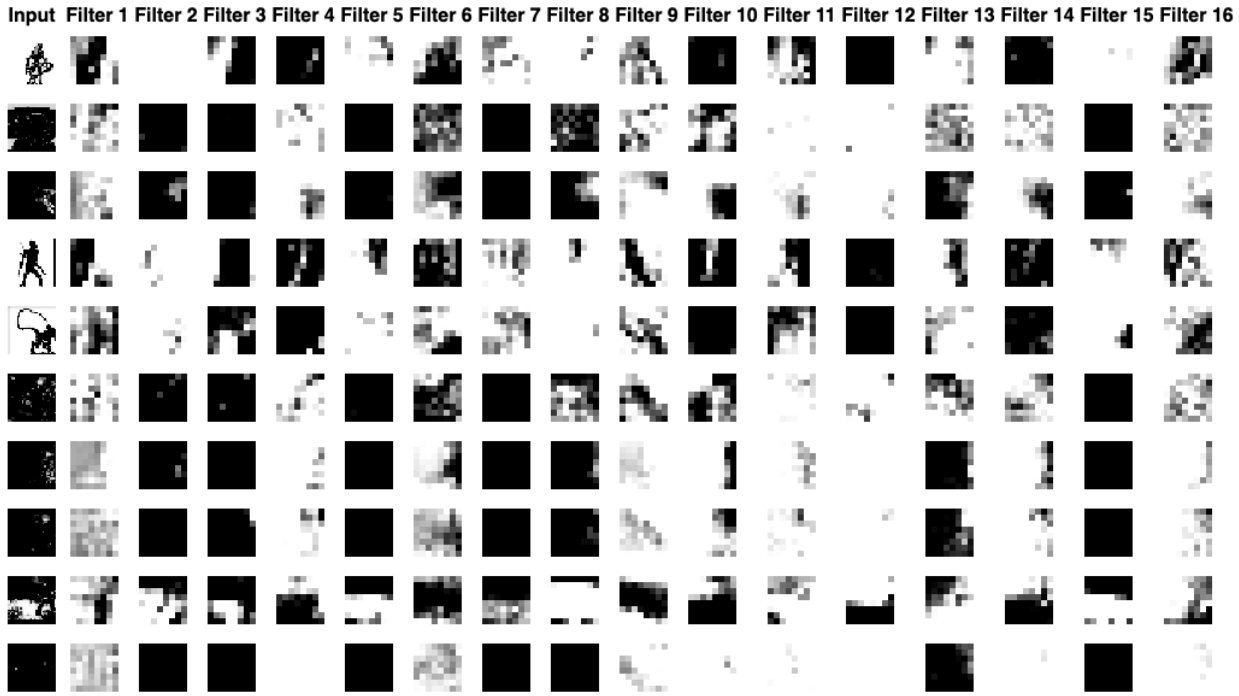
Looks like our model is doing better at "understanding" what an Alien is compared to a Predator. It is also biased towards predicting anything is an Alien.

What filters did our model learn in the first convolutional layer?



You can see that the visualization of the filters contains much fuzzier edges. We can see the same

thing happening in the second layer:



One interesting observation from this layer is the fact that most of the filters now contain highly contrasting patches. Seems like our model did learn something.

Conclusion

You just built your first Convolutional Neural Network model and can now use it to predict whether an image contains an Alien or a Predator. You also learned how to:

- Load images from remote URLs in the browser
- Transform (turn to greyscale, resize and rescale) image pixel data into Tensors
- Build a simple Deep Convolutional Neural Network from scratch using TensorFlow.js
- Visualize filters (what the network learns) of convolutional layers

The methods and techniques for building our model are very general and can be used for both classification and regression tasks when using raw image data. Try them on your own datasets!

Run the complete source code for this tutorial right in your browser[118]

Source code on GitHub[119]

[118] https://codesandbox.io/s/alien-vs-predator-tensorflow-js-classification-ouvt8?fontsize=14
[119] https://github.com/curiousily/alien-vs-predator-tensorflow-js-classification

References

- Keras vs. PyTorch: Alien vs. Predator recognition with transfer learning[120]
- Very Deep Convolutional Networks for Text Classification[121]
- CS231n Convolutional Neural Networks[122]

[120] https://deepsense.ai/keras-vs-pytorch-avp-transfer-learning/
[121] https://arxiv.org/pdf/1606.01781.pdf
[122] https://cs231n.github.io/convolutional-networks/

ToDo List text classification using Embeddings and Deep Neural Networks

> TL;DR Learn how to create a simple ToDo list app in ReactJS and use TensorFlow.js to suggest icons for your tasks based on their name

I know you might be tempted to use your new Machine Learning skills to whatever problem stands in front of you. But I think we can agree that replacing a couple of regex expressions or if/else statements with a complex model is rarely appropriate. I view building software as a way to make our lifes easier. If you can deliver a quick and simple solution with high enough accuracy, do you need Machine Learning? Probably not. It might be counterintuitive to you, but solving a problem using Machine Learning problem starts with deciding whether or not you should use Machine Learning at all!

I am co-creator of a ToDo List & Calendar app called myPoli[123] which helps you achieve your life goals and have fun along the way! One of our goals is to allow our users to customize their tasks to their liking. We use **colors** and **icons** for that. Another goal of ours is to make the app super easy to use.

We allow our users to choose from a wide variety of icons and colors when creating a new Quest (task). But The Paradox of Choice[124] suggests we might be doing them a disservice. I've experienced the blank stare for a couple of seconds when opening the icon picker myself. I also noticed that I use the same icons for similar Quests, but still a large number of different icons.

Here's what you'll learn:

- Build a simple ToDo app using ReactJS
- Preprocess text data
- Use a pre-trained model to create embeddings from text
- Save/load your model
- Build a Deep Neural Network for text classification
- Integrate your model with the ToDo app and deploy it

Can we decrease the cognitive load of our users (help them make fewer decisions) by suggesting an icon based on the ToDo name? Can we do it using Machine Learning?

[123] https://www.mypoli.fun/
[124] https://www.amazon.com/Paradox-Choice-More-Less-Revised-ebook/dp/B000TDGGVU

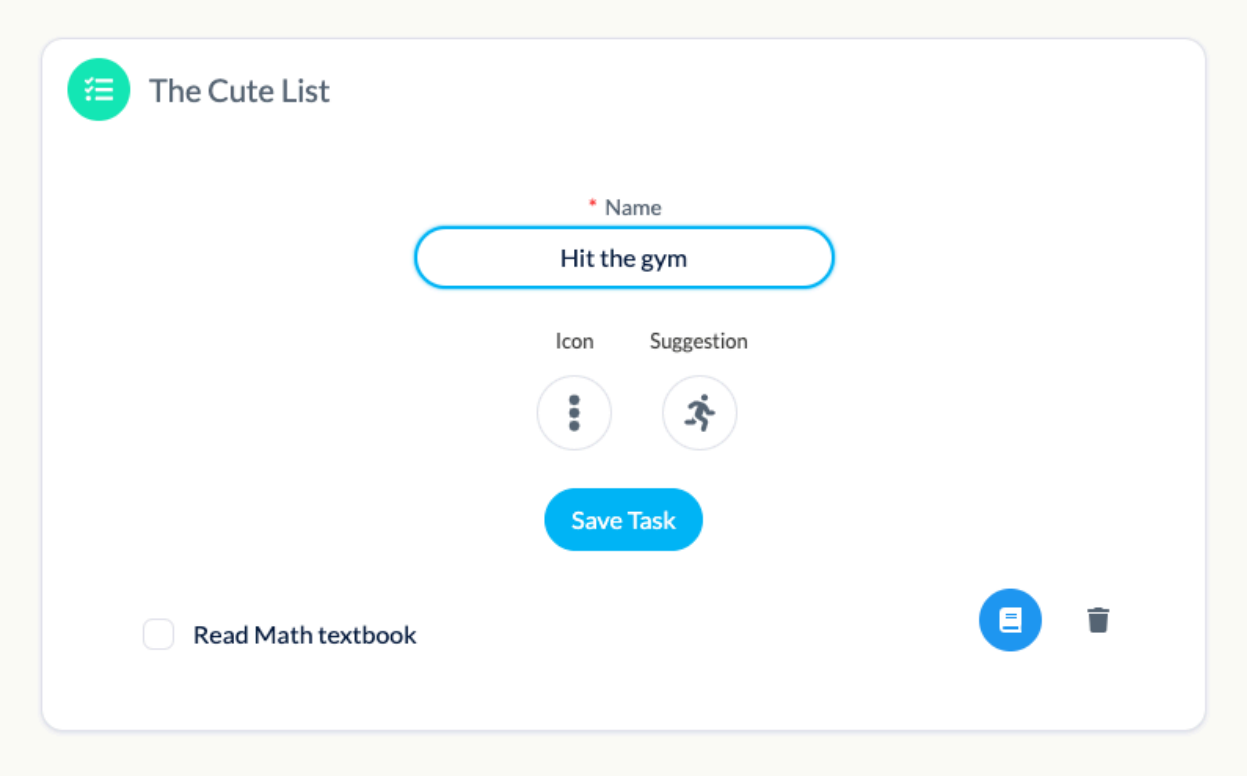
Run the complete source code for this tutorial right in your browser[125]

Source code on GitHub[126]

Live demo of the Cute List app[127]

ToDo app in ReactJS

To answer our question, we'll develop a simple prototype using ReactJS[128] and TensorFlow.js[129] and deploy it using Netlify[130].



You can view a live demo of the Cute List app[131] hosted on Netlify.

While this is not an introduction (in any way) to ReactJS, I want to show you a part of the NewTask component:

[125] https://codesandbox.io/s/todo-list-icon-classification-with-tensorflow-js-9nt78?fontsize=14
[126] https://github.com/curiousily/todo-list-icon-classification-with-tensorflow-js
[127] https://cute-list.netlify.com/
[128] https://reactjs.org/
[129] https://www.tensorflow.org/js
[130] https://www.netlify.com/
[131] https://cute-list.netlify.com/

```
const CONFIDENCE_THRESHOLD = 0.65;

const NewTask = ({ onSaveTask, model, encoder }) => {
  const [task, setTask] = useState({
    name: "",
    icon: null
  });

  const [errors, setErrors] = useState([]);

  const [suggestedIcon, setSuggestedIcon] = useState(null);

  const [typeTimeout, setTypeTimeout] = useState(null);

  const handleNameChange = async e => {
    const taskName = e.target.value;

    setTask({
      ...task,
      name: taskName
    });

    setErrors([]);

    if (typeTimeout) {
      clearTimeout(typeTimeout);
    }

    setTypeTimeout(
      setTimeout(async () => {
        const predictedIcon = await suggestIcon(
          model,
          encoder,
          taskName,
          CONFIDENCE_THRESHOLD
        );
        setSuggestedIcon(predictedIcon);
      }, 400)
    );
  };
...
```

Every time the input (task name) is changed, the function `handleNameChange()` is called with the

new text. Here, we have an opportunity to suggest an icon based on that text.

We're using a function called `suggestIcon()` to decide which icon should be used based on the current task name. Note that we're all throttling our predictions - we make suggestions only after the user has stopped writing for 400 milliseconds.

We're also using a confidence threshold. We're not making suggestions if the predictions are below the required certainty of 65%.

Data

Our data comes from a fictional ToDo list app. ToDos look like this:

```
[
  { text: "Workout 15 minutes", icon: "RUN" },
  { text: "Read book", icon: "BOOK" }
];
```

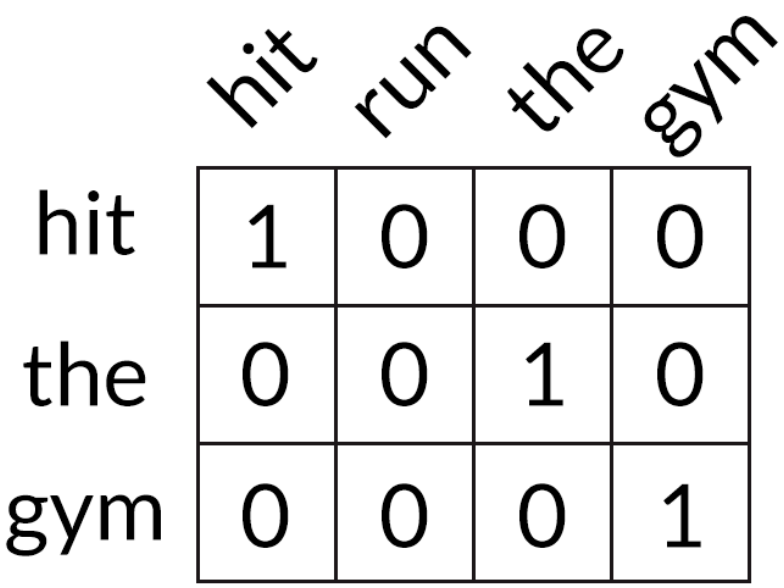
We have around 160 examples.

Embeddings

Similar to representing images, storing text is done by converting characters into numbers. Those numbers are stored in vectors and used by Machine Learning models. There are several ways to turn strings into vectors:

One-hot encoding

We've seen one-hot encoding when classifying images. Each unique word in the sentence is represented with a zero vector (with length the number of unique words in the sentence) and one at the chosen index for the word.



One-hot encoding

Word Embeddings

Another way to encode words into numbers is to use embeddings. They encode similar words with similar floating-point numbers. More importantly, this encoding is learned from the text itself. You can specify the dimensions (usually between 8 and 1024) as the number of parameters. Higher dimensions can capture similarities between words better.

hit	0.52	0.13	0.87	0.08
the	0.11	0.34	0.14	0.09
gym	0.78	0.99	0.22	0.42

4-dimensional embedding

Embedding ToDos

For us, the power of embeddings lies within the similarity scores between words. We can extend that to getting similarity scores between whole sentences. Let's try that with some ToDos:

```
const ToDos = [
  "Hit the gym",
  "Go for a run",
  "Study Math",
  "Watch Biology lectures",
  "Date with Michele",
  "Have dinner with Pam"
];
```

Here, we'll use a shortcut - a pre-trained model on a much larger corpus (set of sentences). Pre-trained models are used in a variety of subfields in Machine Learning, especially Computer Vision (Convolutional Neural Networks) and Natural Language Processing.

In particular, we'll use the Universal Sentence Encoder Lite (USE)[132] that encodes into *512* embeddings and uses a vocabulary of 8,000 words. An additional benefit of the model is that it is trained on short sentences/phrases (just like ToDo items):

[132] https://github.com/tensorflow/tfjs-models/tree/master/universal-sentence-encoder

> The model is trained and optimized for greater-than-word length text, such as sentences, phrases or short paragraphs. It is trained on a variety of data sources and a variety of tasks with the aim of dynamically accommodating a wide variety of natural language understanding tasks.

Let's see how we can use the model[133] to embed the first ToDo in the list:

```
import * as use from "@tensorflow-models/universal-sentence-encoder";

const model = await use.load();

const todoEmbedding = await model.embed(ToDos[0]);
console.log(todoEmbedding.shape);
```

```
[1, 512];
```

One sentence with 512 dimensions (embeddings). Let's have a look at some of the values:

```
console.log(todoEmbedding.dataSync());
```

```
Float32Array {0: -0.052551645785570145, 1: -0.011542949825525284 ...}
```

How can we use this to calculate the similarity between two ToDo items:

```
const similarityScore = async (sentenceAIndex, sentenceBIndex, embeddings) => {
  const sentenceAEmbeddings = embeddings.slice([sentenceAIndex, 0], [1]);
  const sentenceBEmbeddings = embeddings.slice([sentenceBIndex, 0], [1]);
  const sentenceATranspose = false;
  const sentenceBTransepose = true;
  const scoreData = await sentenceAEmbeddings
    .matMul(sentenceBEmbeddings, sentenceATranspose, sentenceBTransepose)
    .data();

  return scoreData[0];
};
```

We start by extracting the matrices representing the embeddings for each exercise and multiply them. The resulting Tensor is a scalar value in the 0-1 range.

Let's find the similarity score of the first pair of ToDos:

[133] https://www.npmjs.com/package/@tensorflow-models/universal-sentence-encoder

```
const todoEmbeddings = await model.embed(ToDos);
const firstPairScore = await similarityScore(0, 1, todoEmbeddings);
console.log(`${ToDos[0]}\n${ToDos[1]}\nsimilarity: ${firstPairScore}`);
```

```
"Hit the gym"
"Go for a run"
similarity: 0.5848015546798706
```

Those two can be put in a "Workout" or "Sports" category. Our model thinks they are relatively similar, too. That's a good start! Let's look at a pair that should not be so similar:

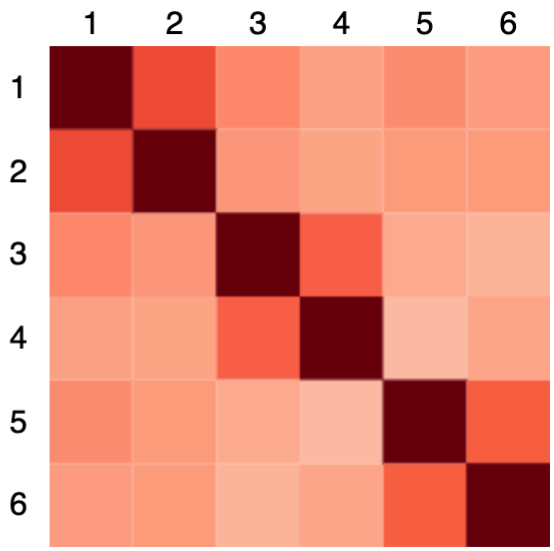
```
const firstThirdScore = await similarityScore(0, 2, todoEmbeddings);
console.log(`${ToDos[0]}\n${ToDos[2]}\nsimilarity: ${firstThirdScore}`);
```

```
Hit the gym
Study Math
similarity: 0.39764219522476196
```

Much lower score. That's somewhat impressive! Note that those ToDos contain only 2-3 words each.

Let's have a look at the similarity matrix for each pair of ToDos:

1) Hit the gym
2) Go for a run
3) Study Math
4) Watch Biology lectures
5) Date with Michele
6) Have dinner with Pam



The pre-trained model seems to capture the similarities pretty well. We have one piece of the puzzle. But how can we use this to suggest icons for ToDos?

Suggesting icons for ToDos

We'll build a model that uses the embeddings from the USE and suggest one of two icons for a ToDo. Those icons are **BOOK** and **RUN**.

Data preprocessing

Let's encode our data and extract the embeddings using USE:

```
const encodeData = async (encoder, tasks) => {
  const sentences = tasks.map(t => t.text.toLowerCase());
  const embeddings = await encoder.embed(sentences);
  return embeddings;
};

const xTrain = await encodeData(encoder, trainTasks);
```

Finally, we'll convert the icon name for each ToDo into one-hot encoded vectors:

```
const yTrain = tf.tensor2d(
  trainTasks.map(t => [t.icon === "BOOK" ? 1 : 0, t.icon === "RUN" ? 1 : 0])
);
```

Using Embeddings in your Deep Neural Network

Now that our data is ready we can start training our model. And it's going to be a really simple one:

```
const N_CLASSES = 2;

const model = tf.sequential();

model.add(
  tf.layers.dense({
    inputShape: [xTrain.shape[1]],
    activation: "softmax",
    units: N_CLASSES
  })
);

model.compile({
```

```
  loss: "categoricalCrossentropy",
  optimizer: tf.train.adam(0.001),
  metrics: ["accuracy"]
});
```

We're going to use the embeddings from USE as features for our model. Our training data contains ~160 examples, which is not much, but we have only two classes.

Training

Training is very similar to how we've train models so far:

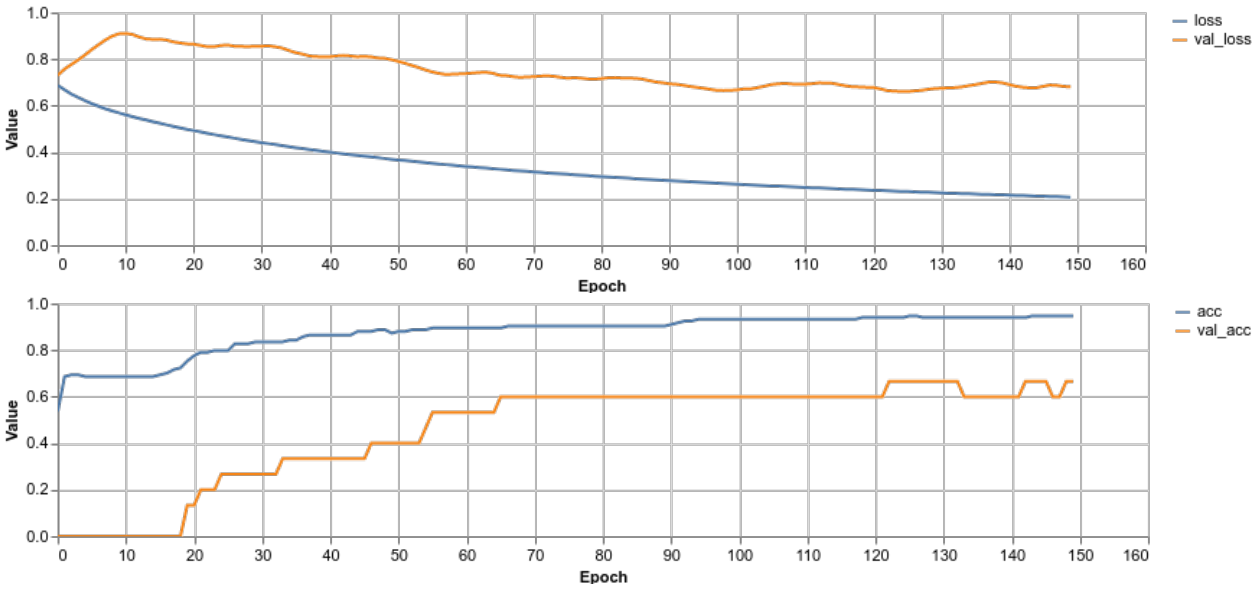
```
const MODEL_NAME = "suggestion-model";

const lossContainer = document.getElementById("loss-cont");

await model.fit(xTrain, yTrain, {
  batchSize: 32,
  validationSplit: 0.1,
  shuffle: true,
  epochs: 150,
  callbacks: tfvis.show.fitCallbacks(
    lossContainer,
    ["loss", "val_loss", "acc", "val_acc"],
    {
      callbacks: ["onEpochEnd"]
    }
  )
});

await model.save(`localstorage://${MODEL_NAME}`);
```

The final line of our code saves the model in Local Storage[134] for later use. That means that we don't have to train our model every time we want to suggest an icon for a ToDo.

Evaluation

We train our model for *150* epochs. Here's what my training progress looks like:

[134] https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage



We hit about 70% accuracy on the validation set.

That would be the end of our analysis if we were doing just that - an analysis. This time, we want to "experience" if the model is doing something useful. Can it suggest good icons for your ToDos?

Recall that we're using the `suggestIcon()` function to that and we can specify how much confident our model should be to make a prediction. Here's how that function is defined:

```
const suggestIcon = async (model, encoder, taskName, threshold) => {
  if (!taskName.trim().includes(" ")) {
    return null;
  }
  const xPredict = await encodeData(encoder, [{ text: taskName }]);

  const prediction = await model.predict(xPredict).data();

  if (prediction[0] > threshold) {
    return "BOOK";
  } else if (prediction[1] > threshold) {
    return "RUN";
  } else {
    return null;
  }
};
```

We start by requiring our task name to include at least one space between characters. We return no prediction when that requirement is not met. We proceed by encoding the task name and using that to make a prediction.

We make the suggestion based on whether or not the threshold is met for the first icon (Book), the second icon (Run) or not met at all.

Deployment

The final step is to deploy your ReactJS app and your model, so it is available for your users. Fortunately, a free and simple way to do that is to use Netlify[135] (I am not affiliated). Have a look at the Deploy React Apps in less than 30 Seconds[136] and learn how to do it.

On a side note, I use Git and GitHub to deploy to Netlify automatically on every commit. Use the "New site from Git" option in your Netlify dashboard or follow the steps from How to deploy a website to Netlify[137].

Conclusion

Congratulations, you've just used a Machine Learning model in a real-world JavaScript app that does something useful - reduces cognitive load and saves time. Here's what you've learned:

- Build a simple ToDo app using ReactJS
- Preprocess text data
- Use a pre-trained model to create embeddings from text
- Save/load your model
- Build a Deep Neural Network for text classification
- Integrate your model with the ToDo app and deploy it

Run the complete source code for this tutorial right in your browser[138]

Source code on GitHub[139]

You might've noticed that the training and using of our model DOES NOT take a central place in our project structure. That's the way it should be when building real-world software. Most of your code should deliver great UX/UI experience and well-tested business logic, at least for now.

You may have many models in your project, but they still deliver specific services that need to be integrated with the rest of the app. A highly accurate Machine Learning model might still be complete trash if it doesn't deliver value to its users.

Live demo of the Cute List app[140]

[135] https://www.netlify.com/
[136] https://www.netlify.com/blog/2016/07/22/deploy-react-apps-in-less-than-30-seconds/
[137] https://medium.com/the-codelog/how-to-deploy-a-website-to-netlify-35274f478144
[138] https://codesandbox.io/s/todo-list-icon-classification-with-tensorflow-js-9nt78?fontsize=14
[139] https://github.com/curiousily/todo-list-icon-classification-with-tensorflow-js
[140] https://cute-list.netlify.com/

References

- Word embeddings with TensorFlow[141]
- Universal Sentence Encoder[142]
- React - A JavaScript library for building user interfaces[143]

[141] https://www.tensorflow.org/tutorials/text/word_embeddings
[142] https://arxiv.org/pdf/1803.11175.pdf
[143] https://reactjs.org/

Burglar alarm system using Object Detection

> TL;DR Learn how to use TensorFlow's Object Detection model (COCO-SSD) to detect intruders from images and webcam feeds

Automated surveillance has always been a goal for a variety of good/bad actors around the globe. With the advance of Machine Learning, this might've become a lot easier.

You're not interested in all that. You have one simple goal. You just want to sleep at night, when far away from home. A simple burglar detection system that notifies you if something is off will do. Your webcam is laying around, waiting patiently. How can you use it?

Here's what you'll learn:

- Build a simple Burglar Alarm app using ReactJS
- Preprocess image and video data
- Use a pre-trained model to detect objects
- Integrate your model with the React app and deploy it

Run the complete source code for this tutorial right in your browser[144]

Source code on GitHub[145]

Live demo of the Burglar Alarm app[146]

Using pre-trained models

The ultimate "Life Hack" when designing and training a Machine Learning model is to NOT DO IT. Turns out, you can do it! Companies with lots of resources (think data and compute power) share some of their trained models. And you can use them for free!

You skip some important and hard steps - figuring out the architecture of your model, finding appropriate hyperparameters (learning rates, momentums, regularizers) and waiting for the training to complete. This can be helpful when prototyping a solution and you want some quick and good results.

[144] https://codesandbox.io/s/burglar-alarm-system-with-tensorflow-js-and-react-ku9s1?fontsize=14
[145] https://github.com/curiousily/burglar-alarm-system-with-tensorflow-js-and-react
[146] https://burglar-alarm.netlify.com/

Object Detection

Our first task is to find people in images/videos. The general problem is known as object detection[147] and deals with detecting different types of objects in images and videos.

One of the largest datasets that include data for our task is Common Objects in Context(COCO)[148]. TensorFlow.js offers a pre-trained COCO-SSD model[149]. SSD stands for Single Shot MultiBox Detection. The model is capable of detecting 90 classes of objects[150].

COCO-SSD Quick Start

Let's take the model for a quick spin. Start by installing the dependencies:

```
yarn add @tensorflow/tfjs @tensorflow-models/coco-ssd
```

Let's load the model:

```
import * as cocoSsd from "@tensorflow-models/coco-ssd";

const model = await cocoSsd.load({ base: "mobilenet_v2" });
```

The base option controls the CNN model[151] that we're going to use for detecting objects. We're using MobileNet v2[152]. It is heavy and slow but provides the best accuracy.

Let's detect some objects on this image:

```
const predictions = await model.detect(image);
```

The detected result looks like this:

[147] https://en.wikipedia.org/wiki/Object_detection
[148] http://cocodataset.org/
[149] https://github.com/tensorflow/tfjs-models/tree/master/coco-ssd
[150] https://github.com/tensorflow/tfjs-models/blob/master/coco-ssd/src/classes.ts
[151] https://en.wikipedia.org/wiki/Convolutional_neural_network
[152] https://arxiv.org/abs/1801.04381

```
[
  {
    bbox: [
      72.00384736061096,
      90.03258740901947,
      158.9742362499237,
      138.44870698451996
    ],
    class: "tv",
    score: 0.9097887277603149
  },
  {
    bbox: [
      -3.011488914489746,
      70.23924046754837,
      440.5377149581909,
      376.6170576810837
    ],
    class: "person",
    score: 0.8925227522850037
  }
];
```

Each detected object has a class, score (how certain our model is) and a bounding box. The bounding box shows the coordinates of the smallest rectangle that we can draw around the object.

That looks cool. Except that one of the classes should be “Evil Snail”.

Finding intruders

We’ll build a simple React app that can detect intruders from 2 sources - predefined image and webcam feed. We’ll display a simple notification when an intruder is detected. You can replace that by sending an email or any other notification method.

The pre-trained model will do most of the work for us. We’ll have to look for persons in the detected classes (when the confidence is high enough, of course) and send a notification.

From image

We already know that the COCO-SSD model works when there’s something to detect on an image. But imagine you’re laying comfortably at the beach in Hawaii and suddenly receive a burglar alarm notification. How frustrating would it be if it is a false one, and you receive those 2-3 times a day?

Let’s start with an image of something strange:

You left your house. All of a sudden, this balloon started floating around. There are no intruders on this image. Luckily, our model thinks the same way. Feel free to try out with your photos.

Remember that you must filter out classes that are not “person” like so:

```
const personDetections = predictions.filter(p => p.class === "person");
```

From webcam stream

Most surveillance systems receive their input from cameras. We’ll do the same thing - use your webcam for surveillance (scared yet?).

Luckily, HTML5 and TensorFlow.js make using your webcam stream easy. Let’s load the webcam stream:

```
const loadCamera = async () => {
  if (
    navigator.mediaDevices.getUserMedia ||
    navigator.mediaDevices.webkitGetUserMedia
  ) {
    const stream = await navigator.mediaDevices.getUserMedia({
      video: true,
      audio: false
    });
    window.stream = stream;
    videoRef.current.srcObject = stream;
  }
};
```

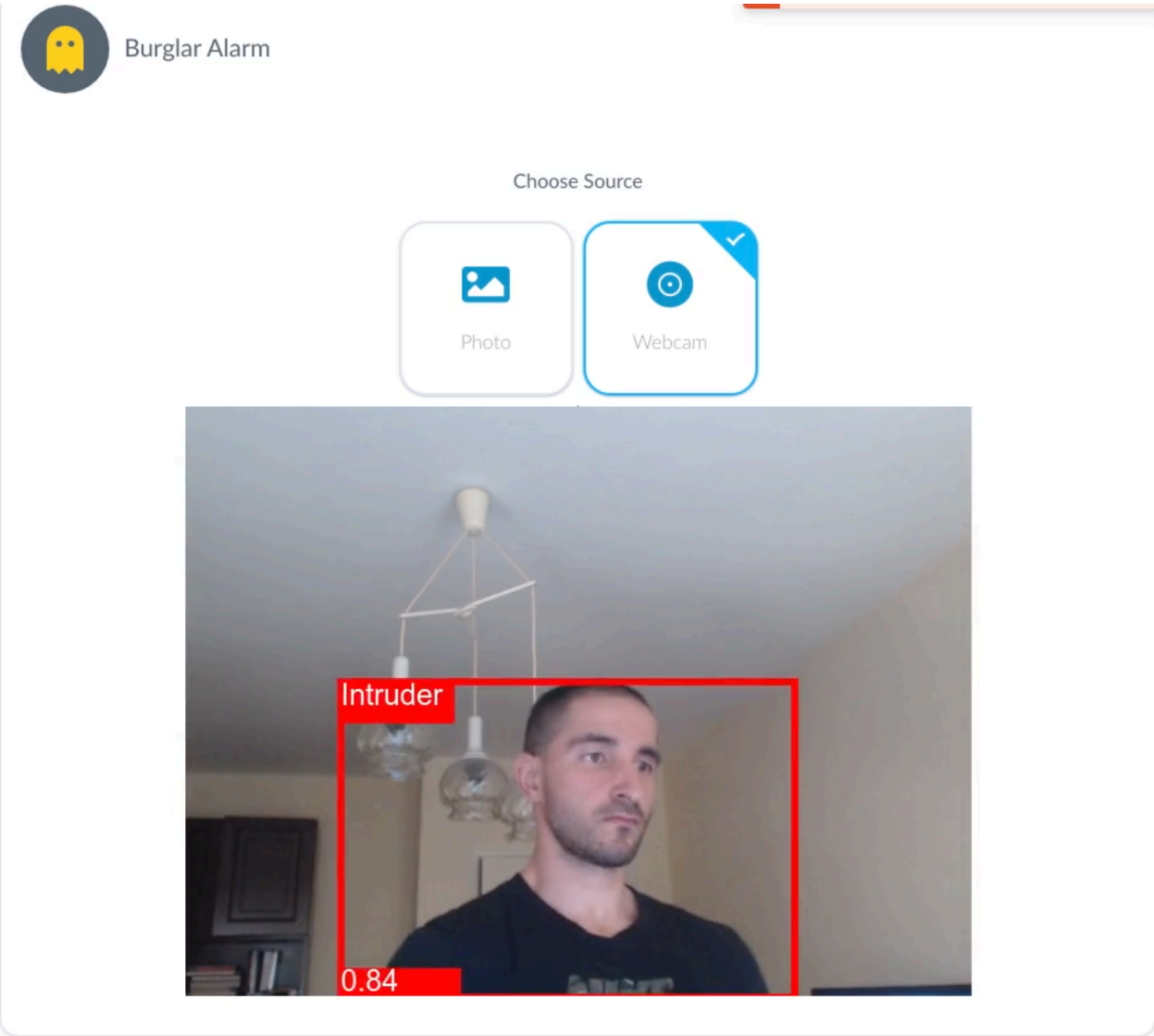
This will request permission to use the webcam. Once you have the permission, our HTML5 video element will show the webcam feed.

Once everything is loaded, the function `detectFromVideoFrame()` is called. Let's have a look at its implementation:

```
const detectFromVideoFrame = async video => {
  try {
    const predictions = await objectDetector.detect(video);

    const personDetections = predictions.filter(p => p.class === "person");

    showDetections(video, personDetections);
    requestAnimationFrame(() => {
      detectFromVideoFrame(video);
    });
  } catch (error) {
    console.log("Couldn't start the webcam");
    console.error(error);
  }
};
```

Essentially, we're chopping the video into images (frames) and detecting persons on each one.

Let's have a look at the final result:



Of course, you might replace the on-screen notification with something more sophisticated. Have a look at the complete source code for the bounding box drawing logic.

Conclusion

Congratulations, you've just used a pre-trained model to detect intruders from images and video. Here's what you've learned:

- Build a simple Burglar Alarm app using ReactJS
- Preprocess image and video data
- Use a pre-trained model to detect objects
- Integrate your model with the React app and deploy it

Imagine that your sweet grandma passes in front of the screen. You wouldn't want to consider her an intruder, right? We'll handle this problem in the next part.

Run the complete source code for this tutorial right in your browser:

Run the complete source code for this tutorial right in your browser[153]

Source code on GitHub[154]

Live demo of the Burglar Alarm app[155]

References

- Object Detection (coco-ssd)[156]
- COCO dataset[157]
- React - A JavaScript library for building user interfaces[158]

[153] https://codesandbox.io/s/burglar-alarm-system-with-tensorflow-js-and-react-ku9s1?fontsize=14
[154] https://github.com/curiousily/burglar-alarm-system-with-tensorflow-js-and-react
[155] https://burglar-alarm.netlify.com/
[156] https://github.com/tensorflow/tfjs-models/tree/master/coco-ssd
[157] http://cocodataset.org/#home
[158] https://reactjs.org/