For the first step, we'll need to build a simple web page that contains a video dev in which we'll render the webcam. We'll build that now. So here's the full page that we'll start with. This page will render a live stream of the webcam. It will also initialize everything you need to start capturing from the webcam and converting that data into tensors, which will then be used to train the network.

Here is where you get the latest tensorflow.js as always

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"> </script>
<script src="webcam.js"></script>
</head>
<body>
  <div>
    <div>
      <video autoplay playsinline muted id="wc" width="224" height="224"></video>
    </div>
  </div>
</body>

<script src="index.js"></script>
</html>
```

This is a link to a webcam.js script. It's provided for you in the repository for this course, and it has been open sourced by Google as a library that manages the webcam in the browser. And in particular for capturing images and converting them to tensors to make your life a little bit easier.

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"> </script>
<script src="webcam.js"></script>
</head>
<body>
  <div>
    <div>
      <video autoplay playsinline muted id="wc" width="224" height="224"></video>
    </div>
  </div>
</body>

<script src="index.js"></script>
</html>
```

This video tag defines a video area on the page, and it's given the id wc that you'll see in a moment.

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"> </script>
<script src="webcam.js"></script>
</head>
<body>
  <div>
    <div>
      <video autoplay playsinline muted id="wc" width="224" height="224"></video>
    </div>
  </div>
</body>

<script src="index.js"></script>
</html>
```

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"> </script>
<script src="webcam.js"></script>
</head>
<body>
  <div>
    <div>
      <video autoplay playsinline muted id="wc" width="224" height="224"></video>
    </div>
  </div>
</body>

<script src="index.js"></script>
</html>
```

This file, index.js, is where we'll write most of our code. You'll create that next and initialize the webcam with it.

So here's how you should start using the index.js file, just keeping it super simple. Declare the mobile net and model variables at the top using code just like this, so that they can be shared across other functions in the script.

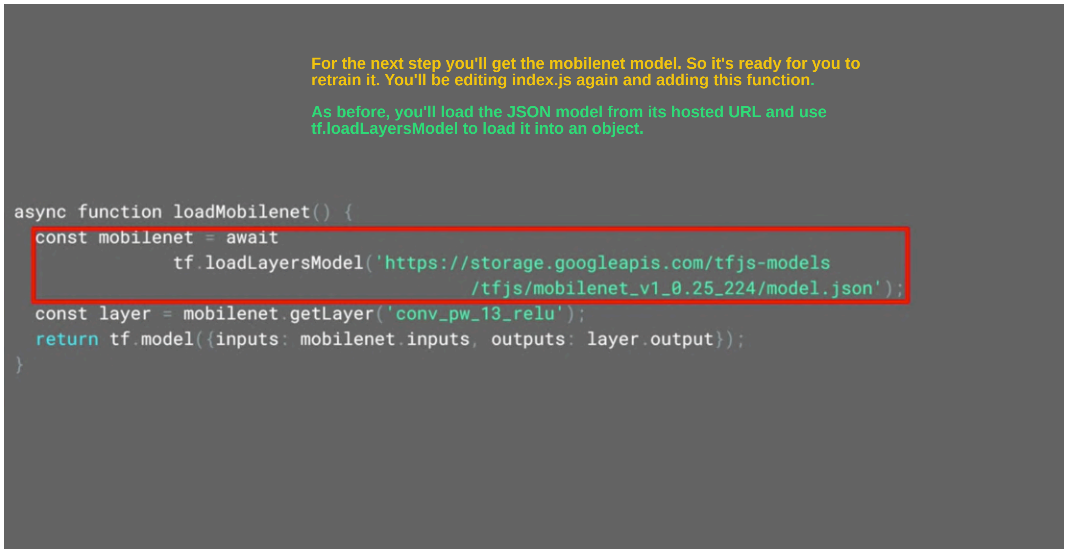
This line creates a const for a webcam object, stored in webcam.js, initializing it by pointing it at the video element in the hosting page that we call wc for webcam.

Remember this webcam class is stored in webcam.js, which is provided for you in the download.

```
let mobilenet;
let model;
const webcam = new Webcam(document.getElementById('wc'));

async function init(){
    await webcam.setup();
}

init();
```

```
let mobilenet;
let model;
const webcam = new Webcam(document.getElementById('wc'));

async function init(){
    await webcam.setup();
}

init();
```

We'll call the init function, which for now, just sets up the webcam by calling webcam.setup.


For the next step you'll get the mobilenet model. So it's ready for you to retrain it. You'll be editing index.js again and adding this function.
As before, you'll load the JSON model from its hosted URL and use tf.loadLayersModel to load it into an object.
async function loadMobilenet() {
const mobilenet = await
tf.loadLayersModel('https://storage.googleapis.com/tfjs-models
/tfjs/mobilenet_v1_0.25_224/model.json');
const layer = mobilenet.getLayer('conv_pw_13_relu');
return tf.model({inputs: mobilenet.inputs, outputs: layer.output});
}


From here, you can now get one of the output layers from the preloaded mobilenet. Remember back to our lesson on transfer learning where you could pick your desired layer and retrain everything underneath that? Well, that's what's happening here. We're selecting the layer called conv p3 13 relu as the one above everything we will freeze.
async function loadMobilenet() {
const mobilenet = await
tf.loadLayersModel('https://storage.googleapis.com/tfjs-models
/tfjs/mobilenet_v1_0.25_224/model.json');
const layer = mobilenet.getLayer('conv_pw_13_relu');
return tf.model({inputs: mobilenet.inputs, outputs: layer.output});
}


We'll then use the tf model class to make a new model and its constructor can take inputs and outputs, which we will set to take the mobilenet inputs, namely the top of the mobilenet and then conv pw 13 relu as output. So that everything beneath that layer will be ignored when we connect a new set of layers to this model.
async function loadMobilenet() {
const mobilenet = await
tf.loadLayersModel('https://storage.googleapis.com/tfjs-models
/tfjs/mobilenet_v1_0.25_224/model.json');
const layer = mobilenet.getLayer('conv_pw_13_relu');
return tf.model({inputs: mobilenet.inputs, outputs: layer.output});
}

In order to make this work, we'll update our init function with a couple of extra lines of code. First of all, we'll call load mobilenet in order to get our model,

```
async function init(){
    await webcam.setup();
    mobilenet = await loadMobilenet();
    tf.tidy(() => mobilenet.predict(webcam.capture()));
}
```

Then, we will initialize the model. The first time around I can take a little time to load all the weights, etc. And so that we don't experience a lag when we want to start training or classifying, I'm going to do a webcam.capture to get a tensor and to ask mobilenet to predict what it sees in that. I don't need to do anything with this, but it does warm up the model for me.

The tf.tidy and throws away any unneeded tensors so that they don't hang around taking up memory.

```
async function init(){
    await webcam.setup();
    mobilenet = await loadMobilenet();
    tf.tidy(() => mobilenet.predict(webcam.capture()));
}
```

Even though we haven't written the code for capturing the data to retrain the network yet, I want to next do the training function, so that you can see how it works a little differently in TensorFlow.js from what you might be used to. Instead of adding a new densely connected set of layers underneath the frozen layers from the original model, we will create a new model. With its input shape being the output shape of the desired mobile net layer.

We then treat this as a separate model that we train. At prediction time, we'll then get a prediction from our truncated mobile net up to the layer that we wanted to give us a set of embeddings.

We'll then pass those embeddings through the new model in order to get a prediction that the new model was trained on. As you can see, it's a little bit different from what you might be used to

Note that unlike Python, this isn't effectively bolted onto the original model. It's an entirely separate one which takes as its input the output from the previous one. So we'll define it like any other model, starting with tf.sequential.

```
async function train() {
  model = tf.sequential({
    layers: [
      tf.layers.flatten({inputShape: mobilenet.outputs[0].shape.slice(1)}),
      tf.layers.dense({ units: 100, activation: 'relu'}),
      tf.layers.dense({ units: 3, activation: 'softmax'})
    ]
  });
}
```

Our first layer will be the flattened output from the mobile net model that we created earlier by truncating the full mobile net.

```
async function train() {
  model = tf.sequential({
    layers: [
      tf.layers.flatten({inputShape: mobilenet.outputs[0].shape.slice(1)}),
      tf.layers.dense({ units: 100, activation: 'relu'}),
      tf.layers.dense({ units: 3, activation: 'softmax'})
    ]
  });
}
```


```
async function train() {
  model = tf.sequential({
    layers: [
      tf.layers.flatten({inputShape: mobilenet.outputs[0].shape.slice(1)}),
      tf.layers.dense({ units: 100, activation: 'relu'}),
      tf.layers.dense({ units: 3, activation: 'softmax'})
    ]
  });
}
```



```
async function train() {
  model = tf.sequential({
    layers: [
      tf.layers.flatten({inputShape: mobilenet.outputs[0].shape.slice(1)}),
      tf.layers.dense({ units: 100, activation: 'relu'}),
      tf.layers.dense({ units: 3, activation: 'softmax'})
    ]
  });
}
```

Here's a snippet of the code that you'll create later to do the inference. You can see that you get a set of embeddings by calling predict.mobilenet, passing in the image.

You then take these embeddings and pass them to the new model to get a prediction photo. That means that when you train this model, you'll be training on the embeddings that you gathered from moblienet

```
const embeddings = mobilenet.predict(img);
const predictions = model.predict(embeddings);
```

We'll start with the code that you need to capture the data that will be used to retrain the network. Here's the first set of changes to your HTML.

This creates three buttons, one for each type of sample that we want to capture. It has three output divs to render the number of samples that have been captured for each, and then another button to start the training. Note how each of the three buttons for gathering data share the same handleButton this as their onClick event handler.

```
<button type="button" id="0" onclick="handleButton(this)" >Rock</button>
<button type="button" id="1" onclick="handleButton(this)" >Paper</button>
<button type="button" id="2" onclick="handleButton(this)" >Scissors</button>
<div id="rocksamples">Rock Samples:</div>
<div id="papersamples">Paper Samples:</div>
<div id="scissorssamples">Scissors Samples:</div>
<button type="button" id="train" onclick="doTraining()" >Train Network</button>
```

```
<button type="button" id="0" onclick="handleButton(this)" >Rock</button>
<button type="button" id="1" onclick="handleButton(this)" >Paper</button>
<button type="button" id="2" onclick="handleButton(this)" >Scissors</button>
<div id="rocksamples">Rock Samples:</div>
<div id="papersamples">Paper Samples:</div>
<div id="scissorssamples">Scissors Samples:</div>
<button type="button" id="train" onclick="doTraining()" >Train Network</button>
```

```html
<button type="button" id="0" onclick="handleButton(this)" >Rock</button>
<button type="button" id="1" onclick="handleButton(this)" >Paper</button>
<button type="button" id="2" onclick="handleButton(this)" >Scissors</button>
<div id="rocksamples">Rock Samples:</div>
<div id="papersamples">Paper Samples:</div>
<div id="scissorssamples">Scissors Samples:</div>
<button type="button" id="train" onclick="doTraining()" >Train Network</button>
```


```html
<button type="button" id="0" onclick="handleButton(this)" >Rock</button>
<button type="button" id="1" onclick="handleButton(this)" >Paper</button>
<button type="button" id="2" onclick="handleButton(this)" >Scissors</button>
<div id="rocksamples">Rock Samples:</div>
<div id="papersamples">Paper Samples:</div>
<div id="scissorssamples">Scissors Samples:</div>
<button type="button" id="train" onclick="doTraining()" >Train Network</button>
```


when the training button is clicked, we'll call the doTraining function

Let's first look at how the handleButton click event works. This will capture a frame from the camera for training. Here's the full code. So let's step through it. Each button had an ID; zero, one, or two. If you remember, when we call the function, we pass the parameter this which is a reference to the button or HTML element, so we call the parameter here elem. We can then switch on its ID.

```javascript
function handleButton(elem){
  switch(elem.id){
  case "0":
    rockSamples++;
    document.getElementById("rocksamples").innerText = "Rock samples:" + rockSamples;
    break;
  case "1":
    paperSamples++;
    document.getElementById("papersamples").innerText = "Paper samples:" + paperSamples;
    break;
  case "2":
    scissorsSamples++;
    document.getElementById("scissorssamples").innerText = "Scissors samples:" + scissorsSamples;
    break;
  }
  label = parseInt(elem.id);
  const img = webcam.capture();
  dataset.addExample(mobilenet.predict(img), label);
}
```

```
function handleButton(elem){
  switch(elem.id){
  case "0":
    rockSamples++;
    document.getElementById("rocksamples").innerText = "Rock samples:" + rockSamples;
    break;
  case "1":
    paperSamples++;
    document.getElementById("papersamples").innerText = "Paper samples:" + paperSamples;
    break;
  case "2":
    scissorsSamples++;
    document.getElementById("scissorssamples").innerText = "Scissors samples:" + scissorsSamples;
    break;
  }
  label = parseInt(elem.id);
  const img = webcam.capture();
  dataset.addExample(mobilenet.predict(img), label);
}
```

```
function handleButton(elem){
  switch(elem.id){
  case "0":
    rockSamples++;
    document.getElementById("rocksamples").innerText = "Rock samples:" + rockSamples;
    break;
  case "1":
    paperSamples++;
    document.getElementById("papersamples").innerText = "Paper samples:" + paperSamples;
    break;
  case "2":
    scissorsSamples++;
    document.getElementById("scissorssamples").innerText = "Scissors samples:" + scissorsSamples;
    break;
  }
  label = parseInt(elem.id);
  const img = webcam.capture();
  dataset.addExample(mobilenet.predict(img), label);
}
```

We'll then extract the label from the ID by converting it into an int.

```
function handleButton(elem){
  switch(elem.id){
  case "0":
    rockSamples++;
    document.getElementById("rocksamples").innerText = "Rock samples:" + rockSamples;
    break;
  case "1":
    paperSamples++;
    document.getElementById("papersamples").innerText = "Paper samples:" + paperSamples;
    break;
  case "2":
    scissorsSamples++;
    document.getElementById("scissorssamples").innerText = "Scissors samples:" + scissorsSamples;
    break;
  }
  label = parseInt(elem.id);
  const img = webcam.capture();
  dataset.addExample(mobilenet.predict(img), label);
}
```

We'll capture the contents of the webcam so we can extract our features.

```
function handleButton(elem){
  switch(elem.id){
  case "0":
    rockSamples++;
    document.getElementById("rocksamples").innerText = "Rock samples:" + rockSamples;
    break;
  case "1":
    paperSamples++;
    document.getElementById("papersamples").innerText = "Paper samples:" + paperSamples;
    break;
  case "2":
    scissorsSamples++;
    document.getElementById("scissorssamples").innerText = "Scissors samples:" + scissorsSamples;
    break;
  }
  label = parseInt(elem.id);
  const img = webcam.capture();
  dataset.addExample(mobilenet.predict(img), label);
}
```

Here's where it gets really interesting. First of all, I haven't introduced the dataset yet, and I'll show the code for that next. But the important thing to notice is that I am not adding the image that I captured from the webcam to the dataset.


```
function handleButton(elem){
  switch(elem.id){
  case "0":
    rockSamples++;
    document.getElementById("rocksamples").innerText = "Rock samples:" + rockSamples;
    break;
  case "1":
    paperSamples++;
    document.getElementById("papersamples").innerText = "Paper samples:" + paperSamples;
    break;
  case "2":
    scissorsSamples++;
    document.getElementById("scissorssamples").innerText = "Scissors samples:" + scissorsSamples;
    break;
  }
  label = parseInt(elem.id);
  const img = webcam.capture();
  dataset.addExample(mobilenet.predict(img), label);
}
```


I'm adding the prediction of that image from my MobileNet. Remember earlier when we said we were doing the transfer learning by removing the bottom layers from the MobileNet, truncating it, so that we just want its output to be the features learned at a higher level. If I then predict on the truncated one, then that's the output that I'll get. So I can train another neural network on those features instead of the raw webcam data and I'll effectively have transfer learning


```
function handleButton(elem){
  switch(elem.id){
  case "0":
    rockSamples++;
    document.getElementById("rocksamples").innerText = "Rock samples:" + rockSamples;
    break;
  case "1":
    paperSamples++;
    document.getElementById("papersamples").innerText = "Paper samples:" + paperSamples;
    break;
  case "2":
    scissorsSamples++;
    document.getElementById("scissorssamples").innerText = "Scissors samples:" + scissorsSamples;
    break;
  }
  label = parseInt(elem.id);
  const img = webcam.capture();
  dataset.addExample(mobilenet.predict(img), label);
}
```


I then also pass the label to the dataset. Note that the label is a zero, a one, or a two. It's not one-hot encoded. But there is a method on the dataset to do the one-hot encoding that we'll see happens just before we train. It just makes this part of the code a little easier to handle.

we'll have to create a script tag to load the dataset class. It's in a file called rps-dataset.js.

```
<script src="rps-dataset.js"></script>
```

Before we use the dataset in our index.js, we have to declare it like this

```
const dataset = new RPSDataset();
```

```
class RPSDataset {
  constructor() {
    this.labels = []
  }
  addExample(example, label) {
    if (this.xs == null) {
      this.xs = tf.keep(example);
      this.labels.push(label);
    } else {
      const oldX = this.xs;
      this.xs = tf.keep(oldX.concat(example, 0));
      this.labels.push(label);
      oldX.dispose();
    }
  }
  encodeLabels(numClasses) {
    ...
  }
}
```

Here's the RPSDataset class found in RPS dataset.js. I've truncated it a little here, I just want to show you how it's structured, and in particular how addExample works.

First of all the labels. As we add new examples to the dataset, we keep track of their labels. That's what the labels array does. So when we initialize the class, I just want to set it to empty.

```
class RPSDataset {
  constructor() {
    this.labels = []
  }
  addExample(example, label) {
    if (this.xs == null) {
      this.xs = tf.keep(example);
      this.labels.push(label);
    } else {
      const oldX = this.xs;
      this.xs = tf.keep(oldX.concat(example, 0));
      this.labels.push(label);
      oldX.dispose();
    }
  }
  encodeLabels(numClasses) {
    ...
  }
}
```

Here's the addexample method that we called earlier. It takes an example and a label.

The example is the output of the prediction for the image from the truncated mobile net.

The label is the values 0, 1, or 2 for rock, paper, and scissors accordingly.

```
class RPSDataset {
  constructor() {
    this.labels = []
  }
  addExample(example, label) {
    if (this.xs == null) {
      this.xs = tf.keep(example);
      this.labels.push(label);
    } else {
      const oldX = this.xs;
      this.xs = tf.keep(oldX.concat(example, 0));
      this.labels.push(label);
      oldX.dispose();
    }
  }
  encodeLabels(numClasses) {
    ...
  }
}
```

So for the first sample, the xs is null. So we set the xs to be the tf.keep for the example, and we push the label into the labels array.

```
class RPSDataset {
  constructor() {
    this.labels = []
  }
  addExample(example, label) {
    if (this.xs == null) {
      this.xs = tf.keep(example);
      this.labels.push(label);
    } else {
      const oldX = this.xs;
      this.xs = tf.keep(oldX.concat(example, 0));
      this.labels.push(label);
      oldX.dispose();
    }
  }
  encodeLabels(numClasses) {
    ...
  }
}
```

So what's this tf.group example code?

Well, remember earlier when we discussed tf.tidy, and how it will throw away all unused tensors. Here, we actually want our tensors to linger around. We'll be calling this function once for every button press. So with tf.keep, we tell TensorFlow that we want to keep this tensor, so please don't throw it away on a tf.tidy.

```
class RPSDataset {
  constructor() {
    this.labels = []
  }
  addExample(example, label) {
    if (this.xs == null) {
      this.xs = tf.keep(example);
      this.labels.push(label);
    } else {
      const oldX = this.xs;
      this.xs = tf.keep(oldX.concat(example, 0));
      this.labels.push(label);
      oldX.dispose();
    }
  }
  encodeLabels(numClasses) {
    ...
  }
}
```

For all subsequent samples, we just append the new example to the old.

We do this by creating its temp variable for the old set of xs called oldX, and we then tf.keep, the concat of the new example to that, and then dispose of the oldX.

We also push the label to the array.

```
class RPSDataset {
  constructor() {
    this.labels = []
  }
  addExample(example, label) {
    if (this.xs == null) {
      this.xs = tf.keep(example);
      this.labels.push(label);
    } else {
      const oldX = this.xs;
      this.xs = tf.keep(oldX.concat(example, 0));
      this.labels.push(label);
      oldX.dispose();
    }
  }
  encodeLabels(numClasses) {
    ...
  }
}
```

If you inspect the JavaScript file, you'll also see encode labels, which takes the array of labels, and one-hot encodes it for training. As one-hot encoding is very memory inefficient, the design was done this way. Keep a list of labels, and only create the much larger list of one-hot encoded ones before you train.

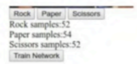
```
async function train() {
  dataset.ys = null;
  dataset.encodeLabels(3);
  model = tf.sequential({
    layers: [
      tf.layers.flatten({inputShape: mobilenet.outputs[0].shape.slice(1)}),
      tf.layers.dense({ units: 100, activation: 'relu'}),
      tf.layers.dense({ units: 3, activation: 'softmax'})
    ]
  });
  const optimizer = tf.train.adam(0.0001);
  model.compile({optimizer: optimizer, loss: 'categoricalCrossentropy'});
  let loss = 0;
  model.fit(dataset.xs, dataset.ys, {
    epochs: 10,
    callbacks: {
      onBatchEnd: async (batch, logs) => {
        loss = logs.loss.toFixed(5);
        console.log('LOSS: ' + loss);
      }
    }
  });
}
```

First of all, we need to one-hot encode the labels in the dataset. Remember there was an array called labels in it. But when we train, we use the y's. So first, we'll set the y's to null, and then we'll call encode labels passing it a three because we have three labels. It will then one-hot encode for us and put the results into dataset.ys.

```
async function train() {
  dataset.ys = null;
  dataset.encodeLabels(3);
  model = tf.sequential({
    layers: [
      tf.layers.flatten({inputShape: mobilenet.outputs[0].shape.slice(1)}),
      tf.layers.dense({ units: 100, activation: 'relu'}),
      tf.layers.dense({ units: 3, activation: 'softmax'})
    ]
  });
  const optimizer = tf.train.adam(0.0001);
  model.compile({optimizer: optimizer, loss: 'categoricalCrossentropy'});
  let loss = 0;
  model.fit(dataset.xs, dataset.ys, {
    epochs: 10,
    callbacks: {
      onBatchEnd: async (batch, logs) => {
        loss = logs.loss.toFixed(5);
        console.log('LOSS: ' + loss);
      }
    }
  });
}
```

```
async function train() {
  dataset.ys = null;
  dataset.encodeLabels(3);
  model = tf.sequential({
    layers: [
      tf.layers.flatten({inputShape: mobilenet.outputs[0].shape.slice(1)}),
      tf.layers.dense({ units: 100, activation: 'relu'}),
      tf.layers.dense({ units: 3, activation: 'softmax'})
    ]
  });
  const optimizer = tf.train.adam(0.0001);
  model.compile({optimizer: optimizer, loss: 'categoricalCrossentropy'});
  let loss = 0;
  model.fit(dataset.xs, dataset.ys, {
    epochs: 10,
    callbacks: {
      onBatchEnd: async (batch, logs) => {
        loss = logs.loss.toFixed(5);
        console.log('LOSS: ' + loss);
      }
    }
  });
}
```

```
async function train() {
  dataset.ys = null;
  dataset.encodeLabels(3);
  model = tf.sequential({
    layers: [
      tf.layers.flatten({inputShape: mobilenet.outputs[0].shape.slice(1)}),
      tf.layers.dense({ units: 100, activation: 'relu'}),
      tf.layers.dense({ units: 3, activation: 'softmax'})
    ]
  });
  const optimizer = tf.train.adam(0.0001);
  model.compile({optimizer: optimizer, loss: 'categoricalCrossentropy'});
  let loss = 0;
  model.fit(dataset.xs, dataset.ys, {
    epochs: 10,
    callbacks: {
      onBatchEnd: async (batch, logs) => {
        loss = logs.loss.toFixed(5);
        console.log('LOSS: ' + loss);
      }
    }
  });
}
```



So now you have the ability to capture data and use that to train a neural network using transfer learning.

You'll see how to poll frames from the webcam, and pass them to the model for inference to see if the model sees rock, paper, or scissors.

For the button that starts the inference, you'll create one and wire it up to the start predictions function in JavaScript. You'll write that function soon.

Do the same for predicting.

```
<div id="dummy">Once training is complete, click 'Start Predicting' to see
predictions, and 'Stop Predicting' to end</div>

<button type="button" id="startPredicting" onclick="startPredicting()" >
    Start Predicting</button>

<button type="button" id="stopPredicting" onclick="stopPredicting()" >
    Stop Predicting</button>

<div id="prediction"></div>
```


```
<div id="dummy">Once training is complete, click 'Start Predicting' to see
predictions, and 'Stop Predicting' to end</div>

<button type="button" id="startPredicting" onclick="startPredicting()" >
    Start Predicting</button>

<button type="button" id="stopPredicting" onclick="stopPredicting()" >
    Stop Predicting</button>

<div id="prediction"></div>
```


These methods will output to the div with predictions that we've just simply named prediction.

```
function startPredicting(){
    isPredicting = true;
    predict();
}
```

```
function startPredicting(){
    isPredicting = true;
    predict();
}
```

```
function stopPredicting(){
    isPredicting = false;
    predict();
}
```

```
while (isPredicting) {
    // Step 1: Get Prediciton

    // Step 2: Evaluate Prediction and Update UI

    // Step 3: Cleanup

  }
```

```
while (isPredicting) {
    // Step 1: Get Prediciton

    // Step 2: Evaluate Prediction and Update UI

    // Step 3: Cleanup

  }
```

```
while (isPredicting) {
    // Step 1: Get Prediciton

    // Step 2: Evaluate Prediction and Update UI

    // Step 3: Cleanup

  }
```

```
while (isPredicting) {
    // Step 1: Get Prediciton

    // Step 2: Evaluate Prediction and Update UI

    // Step 3: Cleanup

  }
```

Here's the code to read a frame from the webcam, use Mobilenet to get the activation, and then get a prediction from that with our retrained model. We'll then arg max this and return it as a one-dimensional Tensor containing the prediction.

As we're dealing with a lot of tensors and memory, and doing it quite frequently, effectively as often as we can, we should tidy up to prevent memory leaks and tf.tidy does that.

```
const predictedClass = tf.tidy(() => {
  const img = webcam.capture();
  const activation = mobilenet.predict(img);
  const predictions = model.predict(activation);
  return predictions.as1D().argMax();
});
```

```
const predictedClass = tf.tidy(() => {
  const img = webcam.capture();
  const activation = mobilenet.predict(img);
  const predictions = model.predict(activation);
  return predictions.as1D().argMax();
});
```

We then call webcam.capture and pass the results to IMG.

```
const predictedClass = tf.tidy(() => {
  const img = webcam.capture();
  const activation = mobilenet.predict(img);
  const predictions = model.predict(activation);
  return predictions.as1D().argMax();
});
```

If you remember our truncated Mobilenet with the bottom layers removed so we could just see the activated convolutions, will pass the image to that to get it set of activations.

```
const predictedClass = tf.tidy(() => {
  const img = webcam.capture();
  const activation = mobilenet.predict(img);
  const predictions = model.predict(activation);
  return predictions.as1D().argMax();
});
```

Now, we can pass that to the model which was trained on these activations for rock paper and scissors classes to get a prediction back. This will then be one heart encoded with results for each of the three classes as probabilities.

```
const predictedClass = tf.tidy(() => {
  const img = webcam.capture();
  const activation = mobilenet.predict(img);
  const predictions = model.predict(activation);
  return predictions.as1D().argMax();
});
```

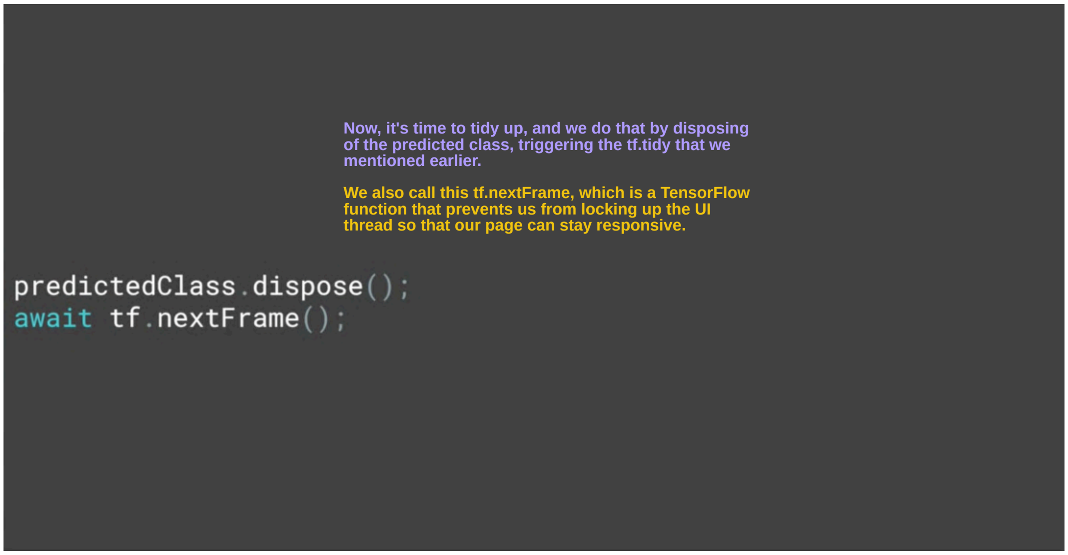
So we arg max that to turn it into a value of zero, one, or two to return it as a classification.

```
const classId = (await predictedClass.data())[0];
var predictionText = "";
switch(classId){
    case 0:
        predictionText = "I see Rock";
        break;
    case 1:
        predictionText = "I see Paper";
        break;
    case 2:
        predictionText = "I see Scissors";
        break;
}
document.getElementById("prediction").innerText = predictionText;
```

First of all, the code we saw previously is called so we can get a class ID from what the webcam sees.

```
const classId = (await predictedClass.data())[0];
var predictionText = "";
switch(classId){
    case 0:
        predictionText = "I see Rock";
        break;
    case 1:
        predictionText = "I see Paper";
        break;
    case 2:
        predictionText = "I see Scissors";
        break;
}
document.getElementById("prediction").innerText = predictionText;
```

```
const classId = (await predictedClass.data())[0];
var predictionText = "";
switch(classId){
    case 0:
        predictionText = "I see Rock";
        break;
    case 1:
        predictionText = "I see Paper";
        break;
    case 2:
        predictionText = "I see Scissors";
        break;
}
document.getElementById("prediction").innerText = predictionText;
```



Rock Paper Scissors

In the next example, we will use a pre-trained MobileNet model to classify hand gestures of Rock, Paper, and Scissors captured by a webcam.

You can use Brackets to open the **index.js** file and take a look at the code. You can find the **index.js** file in the following folder in the GitHub repository for this course:

dlaicourse/TensorFlow Deployment/Course 1 - TensorFlow-JS/Week 4/Examples/

When you launch the **retrain.html** file in the Chrome browser make sure to open the Developer Tools to see the output in the Console.