Transfer Learning

So far, we have trained accurate models on large datasets, and also downloaded a pre-trained model that we used with no training necessary. But what if we cannot find a pre-trained model that does exactly what you need, and what if we do not have a sufficiently large dataset to train a model from scratch? In this case, there is a very helpful technique we can use called transfer learning (https://blogs.nvidia.com/blog/2019/02/07/what-is-transfer-learning/).

With transfer learning, we take a pre-trained model and retrain it on a task that has some overlap with the original training task. A good analogy for this is an artist who is skilled in one medium, such as painting, who wants to learn to practice in another medium, such as charcoal drawing. We can imagine that the skills they learned while painting would be very valuable in learning how to draw with charcoal.

As an example in deep learning, say we have a pre-trained model that is very good at recognizing different types of cars, and we want to train a model to recognize types of motorcycles. A lot of the learnings of the car model would likely be very useful, for instance the ability to recognize headlights and wheels.

Transfer learning is especially powerful when we do not have a large and varied dataset. In this case, a model trained from scratch would likely memorize the training data quickly, but not be able to generalize well to new data. With transfer learning, you can increase your chances of training an accurate and robust model on a small dataset.

Objectives

- Prepare a pretrained model for transfer learning
- Perform transfer learning with your own small dataset on a pretrained model
- Further fine tune the model for even better performance

A Personalized Doggy Door

In our last exercise, we used a pre-trained ImageNet (http://www.image-net.org/) model to let in all dogs, but keep out other animals. In this exercise, we would like to create a doggy door that only lets in a particular dog. In this case, we will make an automatic doggy door for a dog named Bo, the United States First Dog between 2009 and 2017. There are more pictures of Bo in the `data/presidential_doggy_door` folder.

The challenge is that the pre-trained model was not trained to recognize this specific dog, and, we only have 30 pictures of Bo. If we tried to train a model from scratch using those 30 pictures we would experience overfitting and poor generalization. However, if we start with a pre-trained model that is adept at detecting dogs, we can leverage that learning to gain a generalized understanding of Bo using our smaller dataset. We can use transfer learning to solve this challenge.

Downloading the Pretrained Model

The ImageNet pre-trained models (https://keras.io/api/applications/vgg/#vgg16-function) are often good choices for computer vision transfer learning, as they have learned to classify various different types of images. In doing this, they have learned to detect many different types of features (https://developers.google.com/machine-learning/glossary#) that could be valuable in image recognition. Because ImageNet models have learned to detect animals, including dogs, it is especially well suited for this transfer learning task of detecting Bo.

Let us start by downloading the pre-trained model. Again, this is available directly from the Keras library. As we are downloading, there is going to be an important difference. The last layer of an ImageNet model is a dense layer (https://developers.google.com/machine-learning/glossary#dense-layer) of 1000 units, representing the 1000 possible classes in the dataset. In our case, we want it to make a different classification: is this Bo or not? Because we want the classification to be different, we are going to remove the last layer of the model. We can do this by setting the flag `include_top=False` when downloading the model. After removing this top layer, we can add new layers that will yield the type of classification that we want:

In [1]:

```
from tensorflow import keras

base_model = keras.applications.VGG16(
    weights='imagenet',  # Load weights pre-trained on ImageNet.
    input_shape=(224, 224, 3),
    include_top=False)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applica
tions/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58892288/58889256 [==============================] - 0s 0us/step
```

In [2]:

```
base_model.summary()
```

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

Freezing the Base Model

Before we add our new layers onto the pre-trained model (https://developers.google.com/machine-learning/glossary#pre-trained-model), we should take an important step: freezing the model's pre-trained layers. This means that when we train, we will not update the base layers from the pre-trained model. Instead we will only update the new layers that we add on the end for our new classification. We freeze the initial layers because we want to retain the learning achieved from training on the ImageNet dataset. If they were unfrozen at this stage, we would likely destroy this valuable information. There will be an option to unfreeze and train these layers later, in a process called fine-tuning.

Freezing the base layers is as simple as setting trainable on the model to `False`.

In [3]:

```
base_model.trainable = False
```

Adding New Layers

We can now add the new trainable layers to the pre-trained model. They will take the features from the pre-trained layers and turn them into predictions on the new dataset. We will add two layers to the model. First will be a pooling layer like we saw in our earlier convolutional neural network (https://developers.google.com/machine-learning/glossary#convolutional_layer). (If you want a more thorough understanding of the role of pooling layers in CNNs, please read this detailed blog post (https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/#:~:text=A%20pooling%20layer%20is%20a,Convolutional%20Layer)). We then need to add our final layer, which will classify Bo or not Bo. This will be a densely connected layer with one output.

In [4]:

```
inputs = keras.Input(shape=(224, 224, 3))
# Separately from setting trainable on the model, we set training to False
x = base_model(inputs, training=False)
x = keras.layers.GlobalAveragePooling2D()(x)
# A Dense classifier with a single unit (binary classification)
outputs = keras.layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

Let us take a look at the model, now that we have combined the pre-trained model with the new layers.

In [5]:

```
model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
vgg16 (Model)                (None, 7, 7, 512)         14714688
_________________________________________________________________
global_average_pooling2d (Gl (None, 512)               0
_________________________________________________________________
dense (Dense)                (None, 1)                 513
=================================================================
Total params: 14,715,201
Trainable params: 513
Non-trainable params: 14,714,688
_________________________________________________________________
```

Keras gives us a nice summary here, as it shows the vgg16 pre-trained model as one unit, rather than showing all of the internal layers. It is also worth noting that we have many non-trainable parameters as we have frozen the pre-trained model.

Compiling the Model

As with our previous exercises, we need to compile the model with loss and metrics options. We have to make some different choices here. In previous cases we had many categories in our classification problem. As a result, we picked categorical crossentropy for the calculation of our loss. In this case we only have a binary classification problem (Bo or not Bo), and so we will use binary crossentropy (https://www.tensorflow.org/api_docs/python/tf/keras/losses/BinaryCrossentropy). Further detail about the differences between the two can found here (https://gombru.github.io/2018/05/23/cross_entropy_loss/). We will also use binary accuracy instead of traditional accuracy.

By setting `from_logits=True` we inform the loss function (https://gombru.github.io/2018/05/23/cross_entropy_loss/) that the output values are not normalized (e.g. with softmax).

In [6]:

```
# Important to use binary crossentropy and binary accuracy as we now have a binary classif
ication problem
model.compile(loss=keras.losses.BinaryCrossentropy(from_logits=True), metrics=[keras.metri
cs.BinaryAccuracy()])
```

Augmenting the Data

Now that we are dealing with a very small dataset, it is especially important that we augment our data. As before, we will make small modifications to the existing images, which will allow the model to see a wider variety of images to learn from. This will help it learn to recognize new pictures of Bo instead of just memorizing the pictures it trains on.

In [7]:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
# create a data generator
datagen = ImageDataGenerator(
        samplewise_center=True,  # set each sample mean to 0
        rotation_range=10,  # randomly rotate images in the range (degrees, 0 to 180)
        zoom_range = 0.1, # Randomly zoom image
        width_shift_range=0.1,  # randomly shift images horizontally (fraction of total wi
dth)
        height_shift_range=0.1,  # randomly shift images vertically (fraction of total hei
ght)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False) # we don't expect Bo to be upside-down so we will not flip ve
rtically
```

Loading the Data

We have seen datasets in a couple different formats so far. In the MNIST exercise, we were able to download the dataset directly from within the Keras library. For the sign language dataset, the data was in CSV files. For this exercise, we are going to load images directly from folders using Keras' `flow_from_directory` (https://keras.io/api/preprocessing/image/) function. We have set up our directories to help this process go smoothly as our labels are inferred from the folder names. In the `data/presidential_doggy_door` directory, we have train and validation directories, which each have folders for images of Bo and not Bo. In the not_bo directories, we have pictures of other dogs and cats, to teach our model to keep out other pets. Feel free to explore the images to get a sense of our dataset.

Note that flow_from_directory (https://keras.io/api/preprocessing/image/) will also allow us to size our images to match the model: 244x244 pixels with 3 channels.

In [8]:

```
# load and iterate training dataset
train_it = datagen.flow_from_directory('data/presidential_doggy_door/train/',
                                       target_size=(224, 224),
                                       color_mode='rgb',
                                       class_mode='binary',
                                       batch_size=8)
# load and iterate validation dataset
valid_it = datagen.flow_from_directory('data/presidential_doggy_door/valid/',
                                      target_size=(224, 224),
                                      color_mode='rgb',
                                      class_mode='binary',
                                      batch_size=8)
```

```
Found 139 images belonging to 2 classes.
Found 30 images belonging to 2 classes.
```

Training the Model

Time to train our model and see how it does. Recall that when using a data generator, we have to explicitly set the number of `steps_per_epoch`:

In [9]:

```
model.fit(train_it, steps_per_epoch=12, validation_data=valid_it, validation_steps=4, epoc
hs=20)
```

```
Epoch 1/20
12/12 [==============================] - 5s 412ms/step - loss: 0.6772 - binar
y_accuracy: 0.7917 - val_loss: 1.0709 - val_binary_accuracy: 0.7667
Epoch 2/20
12/12 [==============================] - 2s 203ms/step - loss: 0.9221 - binar
y_accuracy: 0.8132 - val_loss: 0.3326 - val_binary_accuracy: 0.8333
Epoch 3/20
12/12 [==============================] - 2s 130ms/step - loss: 0.2240 - binar
y_accuracy: 0.9121 - val_loss: 0.3217 - val_binary_accuracy: 0.9333
Epoch 4/20
12/12 [==============================] - 2s 153ms/step - loss: 0.1864 - binar
y_accuracy: 0.9583 - val_loss: 0.2072 - val_binary_accuracy: 0.9000
Epoch 5/20
12/12 [==============================] - 2s 133ms/step - loss: 0.3625 - binar
y_accuracy: 0.9341 - val_loss: 0.2284 - val_binary_accuracy: 0.9333
Epoch 6/20
12/12 [==============================] - 2s 156ms/step - loss: 0.0389 - binar
y_accuracy: 0.9670 - val_loss: 0.2161 - val_binary_accuracy: 0.9333
Epoch 7/20
12/12 [==============================] - 2s 146ms/step - loss: 0.0587 - binar
y_accuracy: 0.9792 - val_loss: 0.1633 - val_binary_accuracy: 0.9333
Epoch 8/20
12/12 [==============================] - 2s 137ms/step - loss: 0.1248 - binar
y_accuracy: 0.9670 - val_loss: 0.1149 - val_binary_accuracy: 0.9333
Epoch 9/20
12/12 [==============================] - 2s 149ms/step - loss: 0.1001 - binar
y_accuracy: 0.9670 - val_loss: 0.1238 - val_binary_accuracy: 0.9333
Epoch 10/20
12/12 [==============================] - 2s 142ms/step - loss: 0.0364 - binar
y_accuracy: 0.9896 - val_loss: 0.0964 - val_binary_accuracy: 0.9333
Epoch 11/20
12/12 [==============================] - 2s 155ms/step - loss: 0.0192 - binar
y_accuracy: 0.9890 - val_loss: 0.0285 - val_binary_accuracy: 0.9667
Epoch 12/20
12/12 [==============================] - 2s 137ms/step - loss: 0.0709 - binar
y_accuracy: 0.9670 - val_loss: 0.1612 - val_binary_accuracy: 0.9333
Epoch 13/20
12/12 [==============================] - 2s 156ms/step - loss: 0.0539 - binar
y_accuracy: 0.9688 - val_loss: 0.0600 - val_binary_accuracy: 0.9667
Epoch 14/20
12/12 [==============================] - 2s 144ms/step - loss: 0.0017 - binar
y_accuracy: 1.0000 - val_loss: 0.1661 - val_binary_accuracy: 0.9000
Epoch 15/20
12/12 [==============================] - 1s 125ms/step - loss: 0.0144 - binar
y_accuracy: 1.0000 - val_loss: 0.1111 - val_binary_accuracy: 0.9667
Epoch 16/20
12/12 [==============================] - 2s 144ms/step - loss: 0.0151 - binar
y_accuracy: 1.0000 - val_loss: 0.0189 - val_binary_accuracy: 1.0000
Epoch 17/20
12/12 [==============================] - 2s 141ms/step - loss: 0.0209 - binar
y_accuracy: 0.9890 - val_loss: 0.1173 - val_binary_accuracy: 0.9667
Epoch 18/20
12/12 [==============================] - 2s 152ms/step - loss: 0.0030 - binar
y_accuracy: 1.0000 - val_loss: 0.0551 - val_binary_accuracy: 0.9667
Epoch 19/20
12/12 [==============================] - 2s 147ms/step - loss: 0.0033 - binar
y_accuracy: 1.0000 - val_loss: 0.0374 - val_binary_accuracy: 0.9667
```

```
Epoch 20/20
12/12 [==============================] - 2s 151ms/step - loss: 0.0028 - binar
y_accuracy: 1.0000 - val_loss: 0.1400 - val_binary_accuracy: 0.9667
```

Out[9]:

```
<tensorflow.python.keras.callbacks.History at 0x7f45f4cd09b0>
```

Discussion of Results

Both the training and validation accuracy should be quite high. This is a pretty awesome result! We were able to train on a small dataset, but because of the knowledge transferred from the ImageNet model, it was able to achieve high accuracy and generalize well. This means it has a very good sense of Bo and pets who are not Bo.

If you saw some fluctuation in the validation accuracy, that is okay too. We have a technique for improving our model in the next section.

Fine-Tuning the Model

Now that the new layers of the model are trained, we have the option to apply a final trick to improve the model, called fine-tuning (https://developers.google.com/machine-learning/glossary#f). To do this we unfreeze the entire model, and train it again with a very small learning rate (https://developers.google.com/machine-learning/glossary#learning-rate). This will cause the base pre-trained layers to take very small steps and adjust slightly, improving the model by a small amount.

Note that it is important to only do this step after the model with frozen layers has been fully trained. The untrained pooling and classification layers that we added to the model earlier were randomly initialized. This means they needed to be updated quite a lot to correctly classify the images. Through the process of backpropagation (https://developers.google.com/machine-learning/glossary#backpropagation), large initial updates in the last layers would have caused potentially large updates in the pre-trained layers as well. These updates would have destroyed those important pre-trained features. However, now that those final layers are trained and have converged, any updates to the model as a whole will be much smaller (especially with a very small learning rate) and will not destroy the features of the earlier layers.

Let's try unfreezing the pre-trained layers, and then fine tuning the model:

In [10]:

```
# Unfreeze the base model
base_model.trainable = True

# It's important to recompile your model after you make any changes
# to the `trainable` attribute of any inner layer, so that your changes
# are taken into account
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate = .00001),  # Very low lear
ning rate
              loss=keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=[keras.metrics.BinaryAccuracy()])
```

In [11]:

```
model.fit(train_it, steps_per_epoch=12, validation_data=valid_it, validation_steps=4, epochs=10)
```

```
Epoch 1/10
12/12 [==============================] - 8s 668ms/step - loss: 0.1913 - binary_accuracy: 0.9670 - val_loss: 0.0072 - val_binary_accuracy: 1.0000
Epoch 2/10
12/12 [==============================] - 2s 183ms/step - loss: 1.3710e-04 - binary_accuracy: 1.0000 - val_loss: 0.0047 - val_binary_accuracy: 1.0000
Epoch 3/10
12/12 [==============================] - 2s 183ms/step - loss: 6.9046e-04 - binary_accuracy: 1.0000 - val_loss: 0.0052 - val_binary_accuracy: 1.0000
Epoch 4/10
12/12 [==============================] - 2s 173ms/step - loss: 2.1814e-04 - binary_accuracy: 1.0000 - val_loss: 0.0164 - val_binary_accuracy: 1.0000
Epoch 5/10
12/12 [==============================] - 2s 171ms/step - loss: 8.1810e-05 - binary_accuracy: 1.0000 - val_loss: 0.0013 - val_binary_accuracy: 1.0000
Epoch 6/10
12/12 [==============================] - 2s 177ms/step - loss: 1.1718e-04 - binary_accuracy: 1.0000 - val_loss: 0.0170 - val_binary_accuracy: 1.0000
Epoch 7/10
12/12 [==============================] - 2s 192ms/step - loss: 0.0101 - binary_accuracy: 1.0000 - val_loss: 0.0024 - val_binary_accuracy: 1.0000
Epoch 8/10
12/12 [==============================] - 2s 183ms/step - loss: 8.4714e-04 - binary_accuracy: 1.0000 - val_loss: 0.0033 - val_binary_accuracy: 1.0000
Epoch 9/10
12/12 [==============================] - 2s 191ms/step - loss: 6.0931e-05 - binary_accuracy: 1.0000 - val_loss: 0.0112 - val_binary_accuracy: 1.0000
Epoch 10/10
12/12 [==============================] - 2s 173ms/step - loss: 1.8327e-05 - binary_accuracy: 1.0000 - val_loss: 0.0011 - val_binary_accuracy: 1.0000
```

Out[11]:

```
<tensorflow.python.keras.callbacks.History at 0x7f45fd3ba7b8>
```

Examining the Predictions

Now that we have a well-trained model, it is time to create our doggy door for Bo! We can start by looking at the predictions that come from the model. We will preprocess the image in the same way we did for our last doggy door.

In [12]:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from tensorflow.keras.preprocessing import image as image_utils
from tensorflow.keras.applications.imagenet_utils import preprocess_input

def show_image(image_path):
    image = mpimg.imread(image_path)
    plt.imshow(image)

def make_predictions(image_path):
    show_image(image_path)
    image = image_utils.load_img(image_path, target_size=(224, 224))
    image = image_utils.img_to_array(image)
    image = image.reshape(1,224,224,3)
    image = preprocess_input(image)
    preds = model.predict(image)
    return preds
```

Try this out on a couple images to see the predictions:

In [13]:

```
make_predictions('data/presidential_doggy_door/valid/bo/bo_20.jpg')
```

Out[13]:

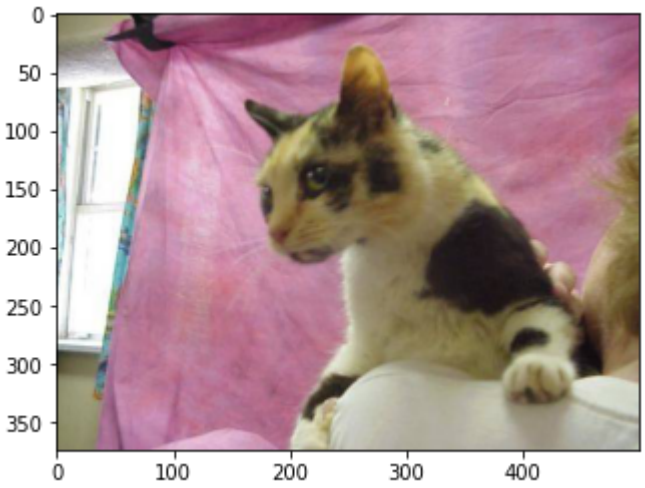
```
array([[-17.810871]], dtype=float32)
```

In [14]:

```
make_predictions('data/presidential_doggy_door/valid/not_bo/121.jpg')
```

Out[14]:

```
array([[24.061687]], dtype=float32)
```



It looks like a negative number prediction means that it is Bo and a positive number prediction means it is something else. We can use this information to have our doggy door only let Bo in!

Exercise: Bo's Doggy Door

Fill in the following code to implement Bo's doggy door:

In [15]:

```
def presidential_doggy_door(image_path):
    preds = make_predictions(image_path)
    if preds < 0:
        print("It's Bo! Let him in!")
    else:
        print("That's not Bo! Stay out!")
```

Solution

Click on the '...' below to see the solution.

```
def presidential_doggy_door(image_path):
    preds = make_predictions(image_path)
    if preds[0] < 0:
        print("It's Bo! Let him in!")
    else:
        print("That's not Bo! Stay out!")
```

Let's try it out!

In [16]:

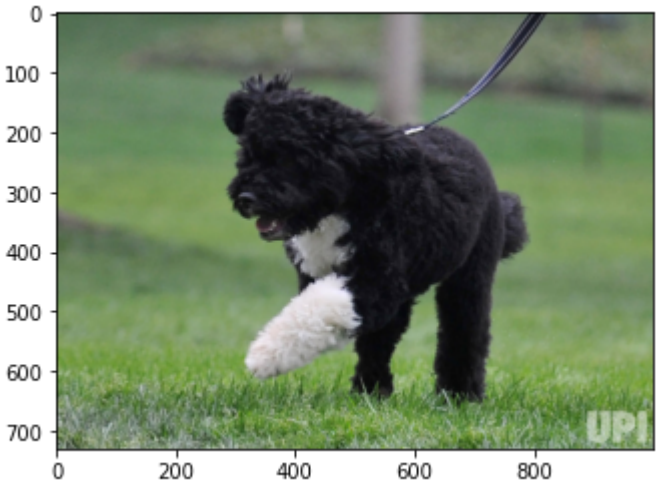
```
presidential_doggy_door('data/presidential_doggy_door/valid/not_bo/131.jpg')
```

```
That's not Bo! Stay out!
```

In [17]:

```
presidential_doggy_door('data/presidential_doggy_door/valid/bo/bo_29.jpg')
```

```
It's Bo! Let him in!
```



Summary

Great work! With transfer learning, you have built a highly accurate model using a very small dataset. This can be an extremely powerful technique, and be the difference between a successful project and one that cannot get off the ground. We hope these techniques can help you out in similar situations in the future!

There is a wealth of helpful resources for transfer learning in the NVIDIA TAO Toolkit (https://developer.nvidia.com/tlt-getting-started).

Clear the Memory

Before moving on, please execute the following cell to clear up the GPU memory.

In [18]:

```
import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

Out[18]:

```
{'status': 'ok', 'restart': True}
```

Next

So far, the focus of this workshop has primarily been on image classification. In the next section, in service of giving you a more well-rounded introduction to deep learning, we are going to switch gears and address working with sequential data, which requires a different approach.