https://www.kaggle.com/learn/deep-learning

https://www.kaggle.com/dansbecker/intro-to-dl-for-computer-vision

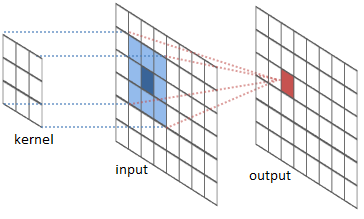
**tensor:** any matrix like object with any dimensions

**convolusion:** small tensors multiplied by small sections of an image to get the specific pattern of the image AKA ‘filters’, eg. Black/white scale, or RGB scale, or brightness. Using the horizontal convolusion or vertical convolution can make the image look like the predator invisibility shield in the Predator movies.

These convolutions are then output to a separate tensor or matrix.

**Intro**

You don't directly choose the numbers to go into your convolutions for deep learning... instead the deep learning technique determines what convolutions will be useful from the data (as part of model-training). We'll come back to how the model does that soon.



But looking closely at convolutions and how they are applied to your image will improve your intuition for these models, how they work, and how to debug them when they don't work.

**Let's get started.**

**Exercises**

We'll use some small utilty functions to visualize raw images and some results of your code. Execute the next cell to load the utility functions.



# Set up code checking

from learntools.core import binder

binder.bind(globals())

from learntools.deep\_learning.exercise\_1 import \*

print("Setup Complete")

WARNING:root:Ignoring repeated attempt to bind to globals

Setup Complete

**Exercise 1**

In the video, you saw a convolution that detected horizontal lines. That convolution shows up again in the code cell below.

Run the cell to see a raw image as well as the output from applying this convolution to the image.



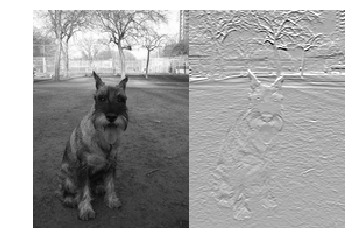
horizontal\_line\_conv = [[1, 1],

[-1, -1]]

# load\_my\_image and visualize\_conv are utility functions provided for this exercise

original\_image = load\_my\_image()

visualize\_conv(original\_image, horizontal\_line\_conv)



Now it's your turn. Instead of a horizontal line detector, you will create a vertical line detector.

**Replace the underscores with numbers to make a vertical line detector and uncomment both lines of code in the cell below. Then run**



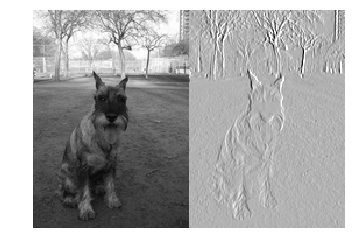
vertical\_line\_conv = [[-1,1],[-1,1]]

​

q\_1.check()

visualize\_conv(original\_image, vertical\_line\_conv)

Correct



If you'd like a hint or the solution, uncomment the appropriate line below.



#q\_1.hint()

#q\_1.solution()

**Exercise 2**

The convolutions you've seen are 2x2. But you could have larger convolutions. They could be 3x3, 4x4, etc. They don't even have to be square. Nothing prevents using a 4x7 convolution.

Compare the number of visual patterns that can be captured by small convolutions. Which of the following is true?

* There are more visual patterns that can be captured by large convolutions
* There are fewer visual patterns that can be captured by large convolutions
* The number of visual patterns that can be captured by large convolutions is the same as the number of visual patterns that can be captured by small convolutions?

Once you think you know the answer, check it by uncommenting and running the line below.



q\_2.solution()

Solution: While any one convolution measures only a single pattern, there are more possible convolutions that can be created with large sizes. So there are also more patterns that can be captured with large convolutions.

For example, it's possible to create a 3x3 convolution that filters for bright pixels with a dark one in the middle. There is no configuration of a 2x2 convolution that would capture this.

On the other hand, anything that can be captured by a 2x2 convolution could also be captured by a 3x3 convolution.

Does this mean powerful models require extremely large convolutions? Not necessarily. In the next lesson, you will see how deep learning models put together many convolutions to capture complex patterns... including patterns to complex to be captured by any single convolution.

**Keep Going**

Now you are ready to [**combine convolutions into powerful models**](https://www.kaggle.com/dansbecker/building-models-from-convolutions)**.** These models are fun to work with, so keep going.

[**Deep Learning Course Home Page**](https://www.kaggle.com/learn/deep-learning)

Convolutions: moving across is moving ‘horizontally’, moving down is moving ‘vertically’ and moving from one output convolution through all output convolutions is ‘channeling’ or the ‘channel’ dimension.

# Intro

At the end of this lesson, you will be able to write TensorFlow and Keras code to use one of the best models in computer vision.

# Lesson

# Sample Code

### Choose Images to Work With

from os.path import join

image\_dir = '../input/dog-breed-identification/train/'

img\_paths = [join(image\_dir, filename) for filename in

['0246f44bb123ce3f91c939861eb97fb7.jpg',

'84728e78632c0910a69d33f82e62638c.jpg',

'8825e914555803f4c67b26593c9d5aff.jpg',

'91a5e8db15bccfb6cfa2df5e8b95ec03.jpg']]

### Function to Read and Prep Images for Modeling

import numpy as np

from tensorflow.python.keras.applications.resnet50 import preprocess\_input

from tensorflow.python.keras.preprocessing.image import load\_img, img\_to\_array

image\_size = 224

def read\_and\_prep\_images(img\_paths, img\_height=image\_size, img\_width=image\_size):

imgs = [load\_img(img\_path, target\_size=(img\_height, img\_width)) for img\_path in img\_paths]

img\_array = np.array([img\_to\_array(img) for img in imgs])

output = preprocess\_input(img\_array)

return(output)

### Create Model with Pre-Trained Weights File. Make Predictions

from tensorflow.python.keras.applications import ResNet50

my\_model = ResNet50(weights='../input/resnet50/resnet50\_weights\_tf\_dim\_ordering\_tf\_kernels.h5')

test\_data = read\_and\_prep\_images(img\_paths)

preds = my\_model.predict(test\_data)

### Visualize Predictions

from learntools.deep\_learning.decode\_predictions import decode\_predictions

from IPython.display import Image, display

most\_likely\_labels = decode\_predictions(preds, top=3, class\_list\_path='../input/resnet50/imagenet\_class\_index.json')

for i, img\_path in enumerate(img\_paths):

display(Image(img\_path))

print(most\_likely\_labels[i])

[('n02097209', 'standard\_schnauzer', 0.5650227), ('n02097047', 'miniature\_schnauzer', 0.31319848), ('n02097130', 'giant\_schnauzer', 0.045194592)]

[('n02092339', 'Weimaraner', 0.9976726), ('n02099849', 'Chesapeake\_Bay\_retriever', 0.0013928412), ('n02109047', 'Great\_Dane', 0.00032280385)]

[('n02105855', 'Shetland\_sheepdog', 0.9133908), ('n02106030', 'collie', 0.08145218), ('n02105056', 'groenendael', 0.0010965357)]

[('n02110627', 'affenpinscher', 0.9366859), ('n02112706', 'Brabancon\_griffon', 0.03435966), ('n02086240', 'Shih-Tzu', 0.011122953)]

# Exercise

Now you are ready to [**use a powerful TensorFlow model**](https://www.kaggle.com/kernels/fork/521452) yourself.

[**Deep Learning Course Home Page**](https://www.kaggle.com/learn/deep-learning)

# Intro

**This is Lesson 4 in the** [**Deep Learning**](https://www.kaggle.com/learn/deep-learning) **track**

At the end of this lesson, you will be able to use transfer learning to build highly accurate computer vision models for your custom purposes, even when you have relatively little data.

# Lesson

# Sample Code

### Specify Model

from tensorflow.python.keras.applications import ResNet50

from tensorflow.python.keras.models import Sequential

from tensorflow.python.keras.layers import Dense, Flatten, GlobalAveragePooling2D

num\_classes = 2

resnet\_weights\_path = '../input/resnet50/resnet50\_weights\_tf\_dim\_ordering\_tf\_kernels\_notop.h5'

my\_new\_model = Sequential()

my\_new\_model.add(ResNet50(include\_top=False, pooling='avg', weights=resnet\_weights\_path))

my\_new\_model.add(Dense(num\_classes, activation='softmax'))

# Say not to train first layer (ResNet) model. It is already trained

my\_new\_model.layers[0].trainable = False

### Compile Model

my\_new\_model.compile(optimizer='sgd', loss='categorical\_crossentropy', metrics=['accuracy'])

### Fit Model

from tensorflow.python.keras.applications.resnet50 import preprocess\_input

from tensorflow.python.keras.preprocessing.image import ImageDataGenerator

image\_size = 224

data\_generator = ImageDataGenerator(preprocessing\_function=preprocess\_input)

train\_generator = data\_generator.flow\_from\_directory(

'../input/urban-and-rural-photos/rural\_and\_urban\_photos/train',

target\_size=(image\_size, image\_size),

batch\_size=24,

class\_mode='categorical')

validation\_generator = data\_generator.flow\_from\_directory(

'../input/urban-and-rural-photos/rural\_and\_urban\_photos/val',

target\_size=(image\_size, image\_size),

class\_mode='categorical')

my\_new\_model.fit\_generator(

train\_generator,

steps\_per\_epoch=3,

validation\_data=validation\_generator,

validation\_steps=1)

Found 72 images belonging to 2 classes.

Found 20 images belonging to 2 classes.

Epoch 1/1

3/3 [==============================] - 21s 7s/step - loss: 0.5654 - acc: 0.7361 - val\_loss: 0.4350 - val\_acc: 0.8500

<tensorflow.python.keras.callbacks.History at 0x7f5f8793fb70>

### Note on Results:

The printed validation accuracy can be meaningfully better than the training accuracy at this stage. This can be puzzling at first.

It occurs because the training accuracy was calculated at multiple points as the network was improving (the numbers in the convolutions were being updated to make the model more accurate). The network was inaccurate when the model saw the first training images, since the weights hadn't been trained/improved much yet. Those first training results were averaged into the measure above.

The validation loss and accuracy measures were calculated **after** the model had gone through all the data. So the network had been fully trained when these scores were calculated.

This isn't a serious issue in practice, and we tend not to worry about it.

# Your Turn

[**Try transfer learning**](https://www.kaggle.com/kernels/fork/532365) yourself.

[**Deep Learning Course Home Page**](https://www.kaggle.com/learn/deep-learning)

In last section:

Dense layer is the sequential layers output as number of classes, in this case 2 for urban and rural. The ‘softmax’ activation turns those outputs into probabilities so that all sum to 1

‘categorical\_crossentropy’ is the same as ‘log loss’ or error

‘sgd’ or stochastic gradient descent minimizes the loss or categorical\_crossentropy loss

The ‘accuracy’ metric will print out the accuracy of correct predictions instead of the loss or error, better interpretability

Preprocessing from ResNet image preprocessing on image data for ImageDataGenerator

For the flow\_from\_directory() it tells where to read the image data, what size to read in of the image, how many images to read in at a time, and that you want to classify these images into different categories



\*\*[Deep Learning Course Home Page](https://www.kaggle.com/learn/deep-learning)\*\*

​

---

​

**Exercise Introduction**

The cameraman who shot our deep learning videos mentioned a problem that we can solve with deep learning.

He offers a service that scans photographs to store them digitally. He uses a machine that quickly scans many photos. But depending on the orientation of the original photo, many images are digitized sideways. He fixes these manually, looking at each photo to determine which ones to rotate.

In this exercise, you will build a model that distinguishes which photos are sideways and which are upright, so an app could automatically rotate each image if necessary.

If you were going to sell this service commercially, you might use a large dataset to train the model. But you'll have great success with even a small dataset. You'll work with a small dataset of dog pictures, half of which are rotated sideways.

Specifying and compiling the model look the same as in the example you've seen. But you'll need to make some changes to fit the model.

**Run the following cell to set up automatic feedback.**



# Set up code checking

from learntools.core import binder

binder.bind(globals())

from learntools.deep\_learning.exercise\_4 import \*

print("Setup Complete")

Setup Complete

**1) Specify the Model**

Since this is your first time, we'll provide some starter code for you to modify. You will probably copy and modify code the first few times you work on your own projects.

There are some important parts left blank in the following code.

Fill in the blanks (marked with \_\_\_\_) and run the cell



from tensorflow.keras.applications import ResNet50

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten, GlobalAveragePooling2D

​

num\_classes = 2

resnet\_weights\_path = '../input/resnet50/resnet50\_weights\_tf\_dim\_ordering\_tf\_kernels\_notop.h5'

​

my\_new\_model = Sequential()

my\_new\_model.add(ResNet50(include\_top=False, pooling='avg', weights=resnet\_weights\_path))

my\_new\_model.add(Dense(num\_classes, activation='softmax'))

​

# Indicate whether the first layer should be trained/changed or not.

my\_new\_model.layers[0].trainable = False

​

step\_1.check()

Correct



# step\_1.hint()

# step\_1.solution()

**2) Compile the Model**

You now compile the model with the following line. Run this cell.



my\_new\_model.compile(optimizer='sgd',

loss='categorical\_crossentropy',

metrics=['accuracy'])

That ran nearly instantaneously. Deep learning models have a reputation for being computationally demanding. Why did that run so quickly?

After thinking about this, check your answer by uncommenting the cell below.



# step\_2.solution()

**3) Review the Compile Step**

You provided three arguments in the compile step.

* optimizer
* loss
* metrics

Which arguments could affect the accuracy of the predictions that come out of the model? After you have your answer, run the cell below to see the solution.



step\_3.solution()

Solution:

* **optimizer** determines how we determine the numerical values that make up the model. So it can affect the resulting model and predictions
* **loss** determines what goal we optimize when determining numerical values in the model. So it can affect the resulting model and predictions
* **metrics** determines only what we print out while the model is being built, but it doesn't affect the model itself.

You may not understand all of this yet. That's totally fine for now. It will become clearer in an upcoming lesson (called A Deeper Understanding of Deep Learning).

**4) Fit Model**

**Your training data is in the directory ../input/dogs-gone-sideways/images/train. The validation data is in ../input/dogs-gone-sideways/images/val**. Use that information when setting up train\_generator and validation\_generator.

You have 220 images of training data and 217 of validation data. For the training generator, we set a batch size of 10. Figure out the appropriate value of steps\_per\_epoch in your fit\_generator call.

Fill in all the blanks (again marked as \_\_\_\_). Then run the cell of code. Watch as your model trains the weights and the accuracy improves.



from tensorflow.keras.applications.resnet50 import preprocess\_input

from tensorflow.keras.preprocessing.image import ImageDataGenerator

​

image\_size = 224

data\_generator = ImageDataGenerator(preprocess\_input)

​

train\_generator = data\_generator.flow\_from\_directory(

directory= '../input/dogs-gone-sideways/images/train',

target\_size=(image\_size, image\_size),

batch\_size=10,

class\_mode='categorical')

​

validation\_generator = data\_generator.flow\_from\_directory(

directory= '../input/dogs-gone-sideways/images/val',

target\_size=(image\_size, image\_size),

class\_mode='categorical')

​

# fit\_stats below saves some statistics describing how model fitting went

# the key role of the following line is how it changes my\_new\_model by fitting to data

fit\_stats = my\_new\_model.fit\_generator(train\_generator,

steps\_per\_epoch=20,

validation\_data= validation\_generator,

validation\_steps=1)

​

step\_4.check()

Found 220 images belonging to 2 classes.

Found 217 images belonging to 2 classes.

Epoch 1/1

3/20 [===>..........................] - ETA: 1s - loss: 0.4791 - acc: 0.7000

/opt/conda/lib/python3.6/site-packages/keras\_preprocessing/image.py:1131: UserWarning: This ImageDataGenerator specifies `featurewise\_center`, but it hasn't been fit on any training data. Fit it first by calling `.fit(numpy\_data)`.

warnings.warn('This ImageDataGenerator specifies '

20/20 [==============================] - 1s 59ms/step - loss: 0.4420 - acc: 0.7600 - val\_loss: 0.1937 - val\_acc: 0.9375

Incorrect: The validation directory should be ../input/dogs-gone-sideways/val. Yours was ../input/dogs-gone-sideways/images/val



step\_4.solution()

Solution:

image\_size = 224

data\_generator = ImageDataGenerator(preprocess\_input)

train\_generator = data\_generator.flow\_from\_directory(

directory=\_\_\_\_,

target\_size=(image\_size, image\_size),

batch\_size=10,

class\_mode='categorical')

validation\_generator = data\_generator.flow\_from\_directory(

directory=\_\_\_\_,

target\_size=(image\_size, image\_size),

class\_mode='categorical')

# fit\_stats below saves some statistics describing how model fitting went

# the key role of the following line is how it changes my\_new\_model by fitting to data

fit\_stats = my\_new\_model.fit\_generator(train\_generator,

steps\_per\_epoch=\_\_\_\_,

validation\_data=\_\_\_\_,

validation\_steps=1)

Can you tell from the results what fraction of the time your model was correct in the validation data?

In the next step, we'll see if we can improve on that.

**Keep Going**

Move on to learn about [**data augmentation**](https://www.kaggle.com/dansbecker/data-augmentation/). It is a clever and easy way to improve your models. Then you'll apply data augmentation to this automatic image rotation problem.

[**Deep Learning Course Home Page**](https://www.kaggle.com/learn/deep-learning)

[**Deep Learning Course Home Page**](https://www.kaggle.com/learn/deep-learning)

# Intro

At the end of this lesson, you will be able to use data augmentation. This trick that makes it seem like you have far more data than you actually have, resulting in even better models..

# Lesson

# Sample Code

We have some model set-up code which you've seen before. It's not our focus for the moment, so it is hidden (but optionally visible by clicking the "code" button below.)

from tensorflow.python.keras.applications import ResNet50

from tensorflow.python.keras.models import Sequential

from tensorflow.python.keras.layers import Dense, Flatten, GlobalAveragePooling2D

num\_classes = 2

resnet\_weights\_path = '../input/resnet50/resnet50\_weights\_tf\_dim\_ordering\_tf\_kernels\_notop.h5'

my\_new\_model = Sequential()

my\_new\_model.add(ResNet50(include\_top=False, pooling='avg', weights=resnet\_weights\_path))

my\_new\_model.add(Dense(num\_classes, activation='softmax'))

# Say not to train first layer (ResNet) model. It is already trained

my\_new\_model.layers[0].trainable = False

my\_new\_model.compile(optimizer='sgd', loss='categorical\_crossentropy', metrics=['accuracy'])

### Fitting a Model With Data Augmentation

!ls ../input/urban-and-rural-photos/rural\_and\_urban\_photos

train val

from tensorflow.python.keras.applications.resnet50 import preprocess\_input

from tensorflow.python.keras.preprocessing.image import ImageDataGenerator

image\_size = 224

data\_generator\_with\_aug = ImageDataGenerator(preprocessing\_function=preprocess\_input,

horizontal\_flip=True,

width\_shift\_range = 0.2,

height\_shift\_range = 0.2)

train\_generator = data\_generator\_with\_aug.flow\_from\_directory(

'../input/urban-and-rural-photos/rural\_and\_urban\_photos/train',

target\_size=(image\_size, image\_size),

batch\_size=24,

class\_mode='categorical')

data\_generator\_no\_aug = ImageDataGenerator(preprocessing\_function=preprocess\_input)

validation\_generator = data\_generator\_no\_aug.flow\_from\_directory(

'../input/urban-and-rural-photos/rural\_and\_urban\_photos/val',

target\_size=(image\_size, image\_size),

class\_mode='categorical')

my\_new\_model.fit\_generator(

train\_generator,

steps\_per\_epoch=3,

epochs=2,

validation\_data=validation\_generator,

validation\_steps=1)

Found 72 images belonging to 2 classes.

Found 20 images belonging to 2 classes.

Epoch 1/2

3/3 [==============================] - 15s 5s/step - loss: 0.9389 - acc: 0.4167 - val\_loss: 1.0767 - val\_acc: 0.6000

Epoch 2/2

3/3 [==============================] - 11s 4s/step - loss: 0.6994 - acc: 0.6806 - val\_loss: 0.5112 - val\_acc: 0.8500

<tensorflow.python.keras.callbacks.History at 0x7faf6c5fd048>

# Exercise

Move on to [**apply data augmentation**](https://www.kaggle.com/kernels/fork/536195) yourself.

[**Deep Learning Course Home Page**](https://www.kaggle.com/learn/deep-learning)

[**Deep Learning Course Home Page**](https://www.kaggle.com/learn/deep-learning)

# Intro

At the end of this lesson, you will understand how stochastic gradient descent and back-propagation are used to set the weights in a deep learning model. These topics are complex, but many experts view them as the most important ideas in deep learning.

# Lesson

**Links Mentioned**

[**ReLU activation function**](https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning)

# Keep Going

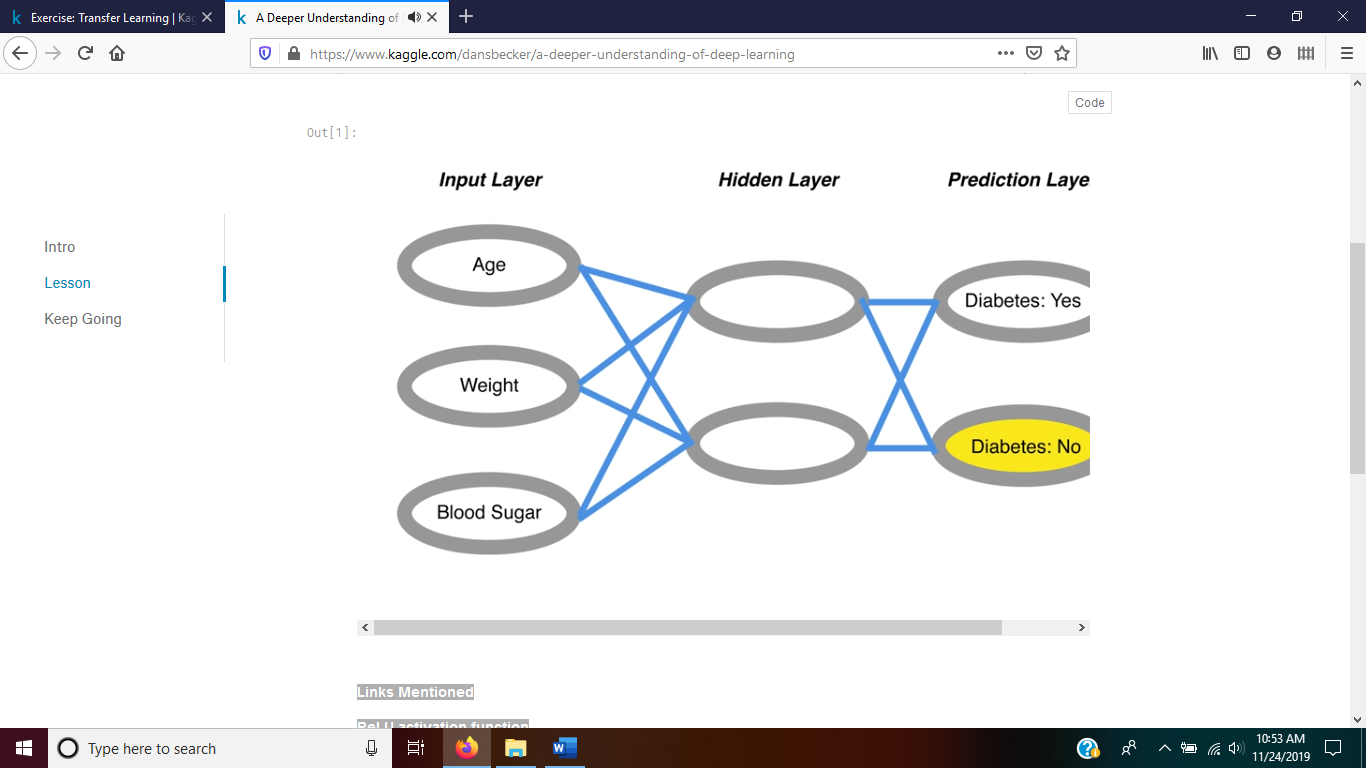
Now you are ready to [**train your own models from scratch**](https://www.kaggle.com/dansbecker/deep-learning-from-scratch)**.**

[**Deep Learning Course Home Page**](https://www.kaggle.com/learn/deep-learning)

Input layer, the ‘raw’ data that is the actual feature values being used

Output layer is the class predicted if final layer, using the weights for each feature,

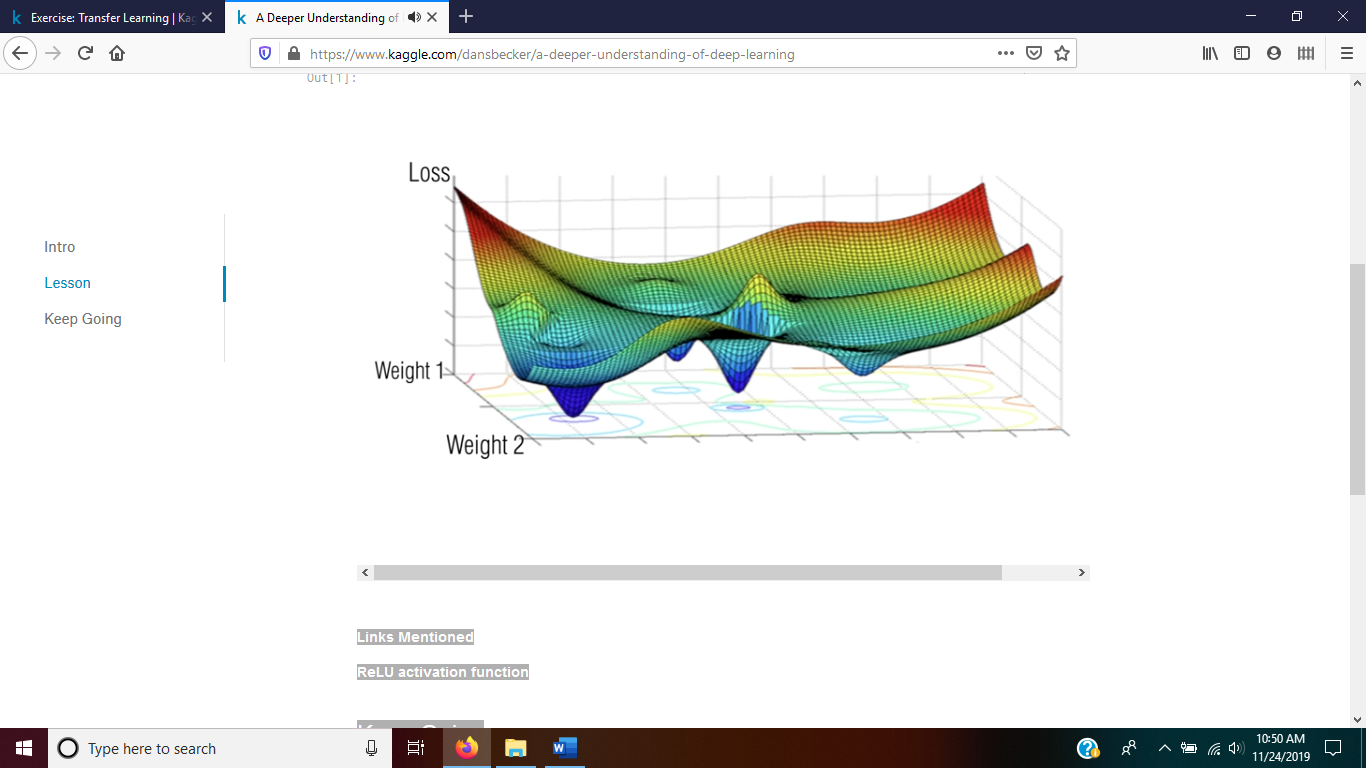
IOW**: linear combination**



**Softmax function** uses this linear combination through that algorithm/function/formula

**Hidden layers** between the input and output layers

**Loss values** are actual-prediction with low meaning good and high implying bad model

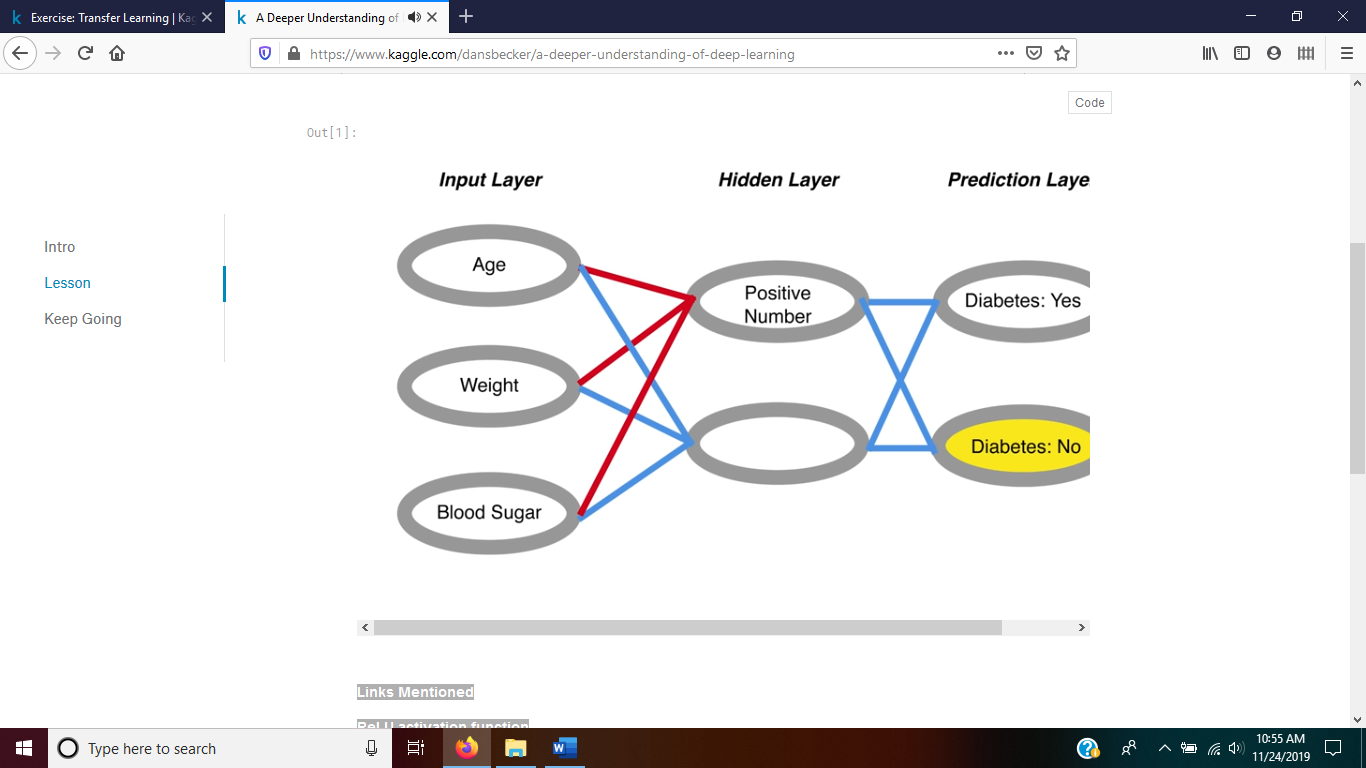


Goal: find the lowest point of loss between 2 features, reality is > 2 dims

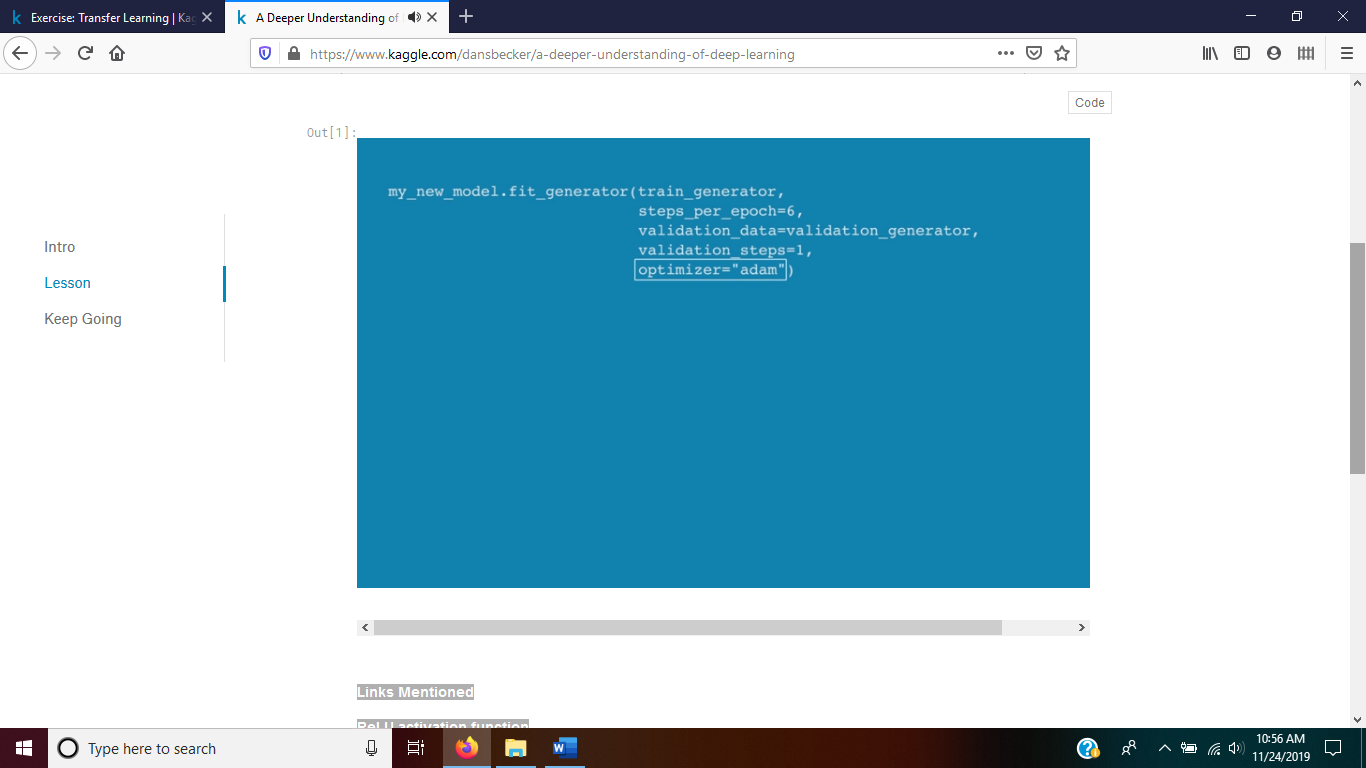
**Backpropagation**: determining if you are going downhill

**Batch size** is the number of rows/instances, or the number of images if using 2D images

**Epochs**: improve weights with each run of the formula or iteration



**Forward propagation**: using the weights, adjusting with each instance, but not too much so model generalizes to other instances in classifying to output layer



**‘adam’** setting is the best for selecting weight adjustments to minimize loss or error

# Intro

**This is Lesson 7 in the** [**Deep Learning**](https://www.kaggle.com/learn/deep-learning) **course**

The models you've built so far have relied on pre-trained models. But they aren't the ideal solution for many use cases. In this lesson, you will learn how to build totally new models.

# Lesson

from IPython.display import YouTubeVideo

YouTubeVideo('YbNE3zhtsoo', width=800, height=450)

# Sample Code

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from tensorflow.python import keras

from tensorflow.python.keras.models import Sequential

from tensorflow.python.keras.layers import Dense, Flatten, Conv2D, Dropout

img\_rows, img\_cols = 28, 28

num\_classes = 10

def data\_prep(raw):

out\_y = keras.utils.to\_categorical(raw.label, num\_classes)

num\_images = raw.shape[0]

x\_as\_array = raw.values[:,1:]

x\_shaped\_array = x\_as\_array.reshape(num\_images, img\_rows, img\_cols, 1)

out\_x = x\_shaped\_array / 255

return out\_x, out\_y

train\_file = "../input/digit-recognizer/train.csv"

raw\_data = pd.read\_csv(train\_file)

x, y = data\_prep(raw\_data)

model = Sequential()

model.add(Conv2D(20, kernel\_size=(3, 3),

activation='relu',

input\_shape=(img\_rows, img\_cols, 1)))

model.add(Conv2D(20, kernel\_size=(3, 3), activation='relu'))

model.add(Flatten())

model.add(Dense(128, activation='relu'))

model.add(Dense(num\_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical\_crossentropy,

optimizer='adam',

metrics=['accuracy'])

model.fit(x, y,

batch\_size=128,

epochs=2,

validation\_split = 0.2)

Train on 33600 samples, validate on 8400 samples

Epoch 1/2

33600/33600 [==============================] - 40s 1ms/step - loss: 0.2303 - acc: 0.9334 - val\_loss: 0.0906 - val\_acc: 0.9727

Epoch 2/2

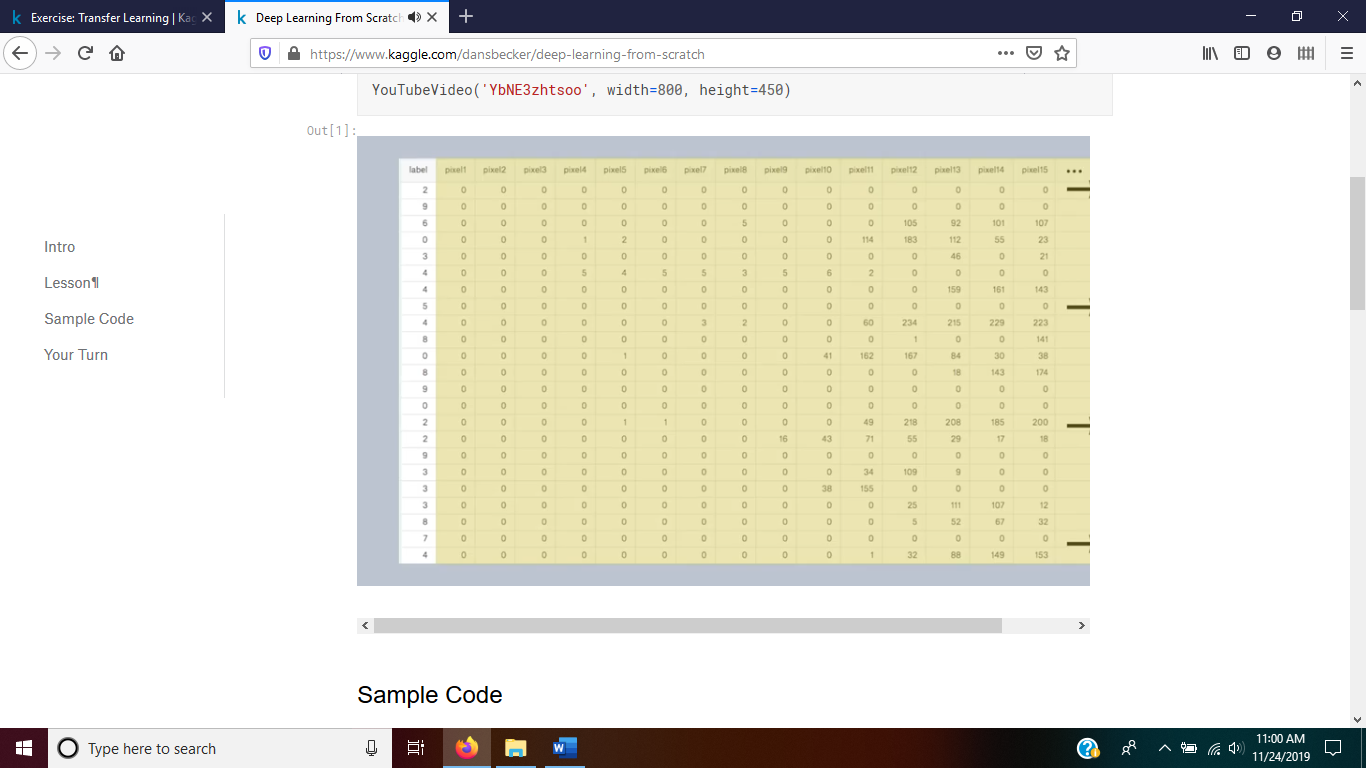
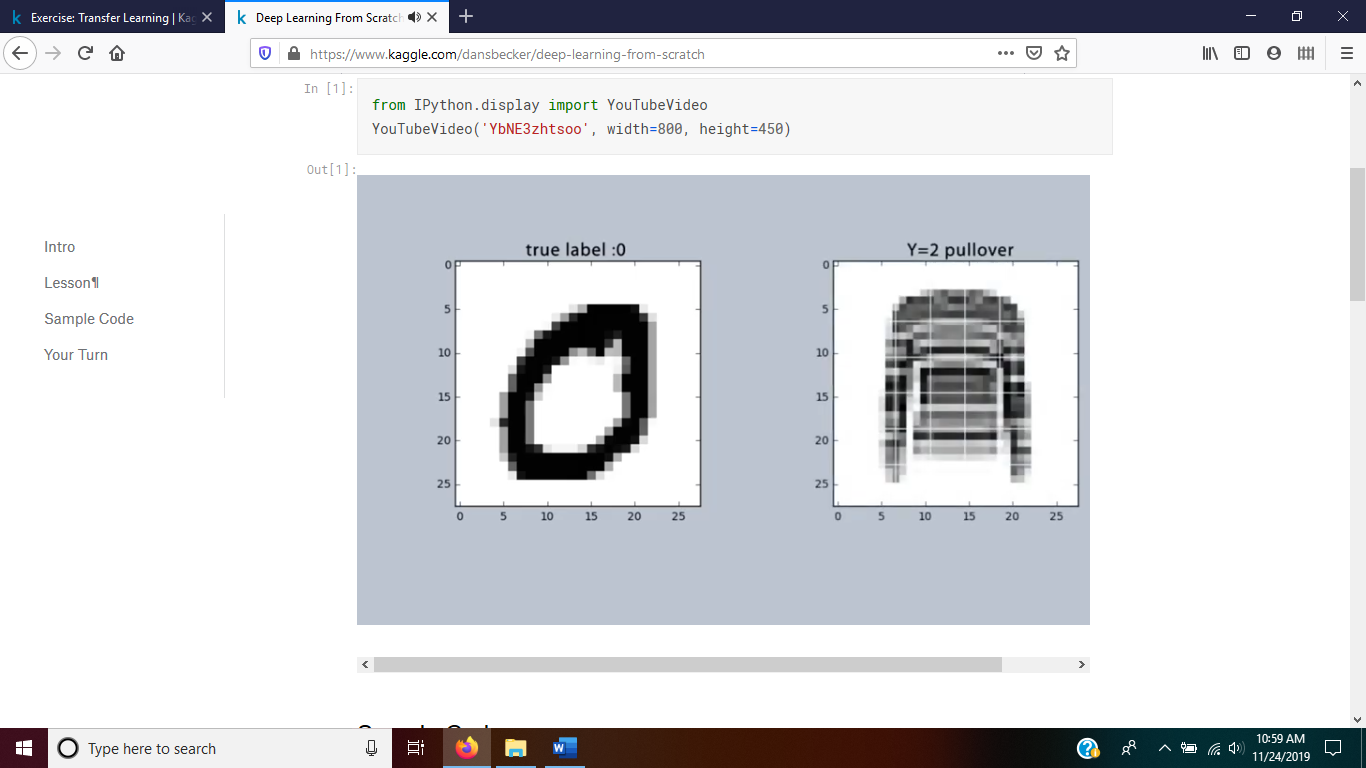
33600/33600 [==============================] - 39s 1ms/step - loss: 0.0632 - acc: 0.9809 - val\_loss: 0.0587 - val\_acc: 0.9817

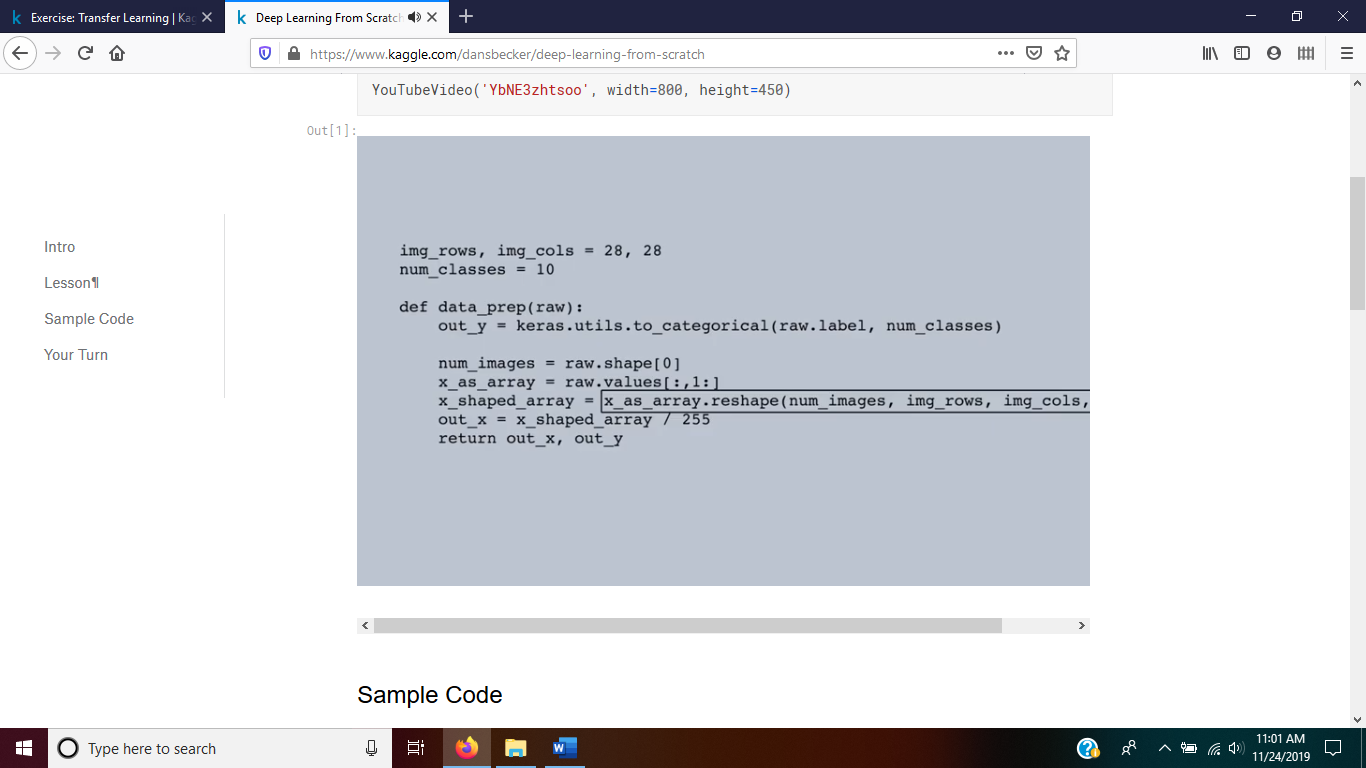
<tensorflow.python.keras.callbacks.History at 0x7fd27e5ebf98>

# Your Turn

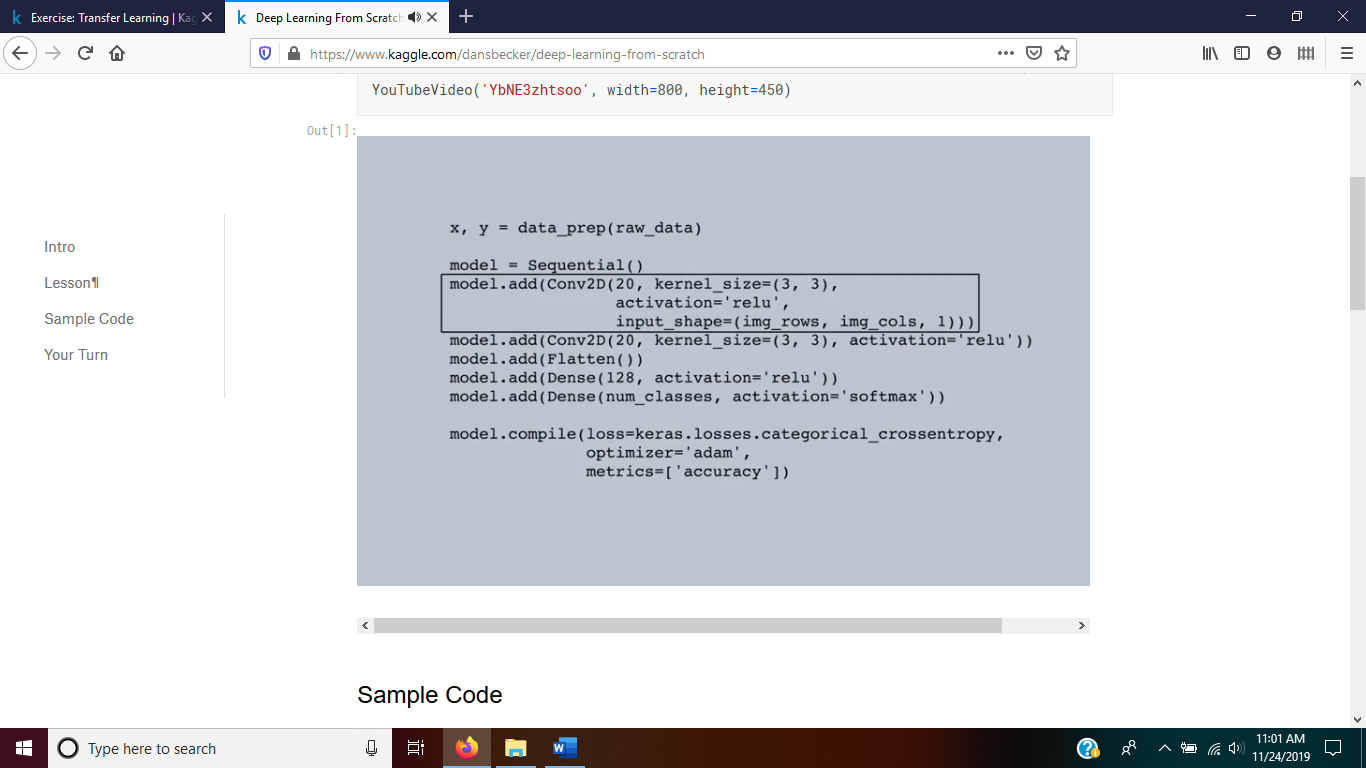
You are ready to [**build your own model**](https://www.kaggle.com/kernels/fork/574269).

[**Deep Learning Course Home Page**](https://www.kaggle.com/learn/deep-learning)

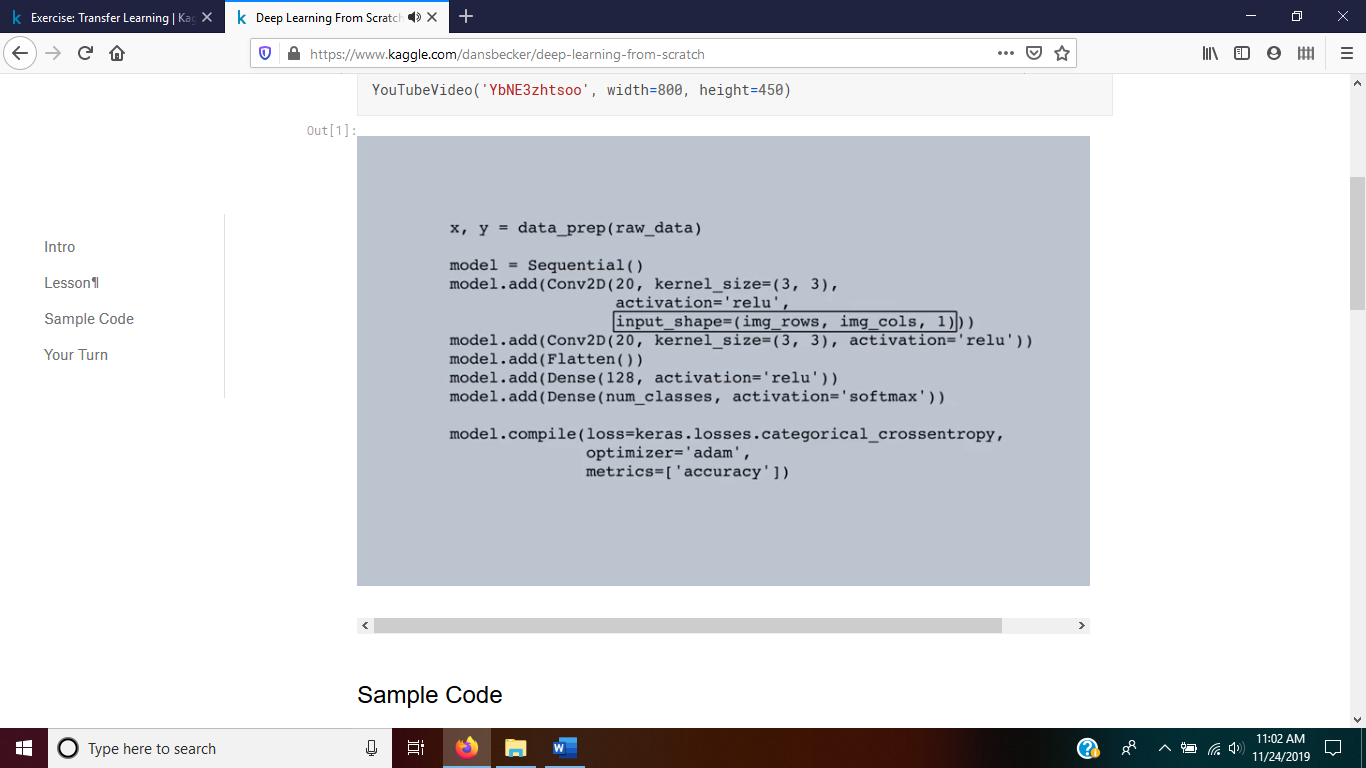
each row is 784 or 28X28 for row/height pixel images



Take everything other than 1st col that was the label, uses one-hot encodeing in code above



Has a conv2D layer and and added layer



The 1st layer needs the shape image specified but not in subsequent layers

Kernel size is the size of the convolution or filter, the 1st number in conv2D() is the number of convolution filters to use



The flatten layer is to flatten the image to a 1D output



Adding a dense() will perform better when placed between the flatten() and final layer, this layer has 128 nodes



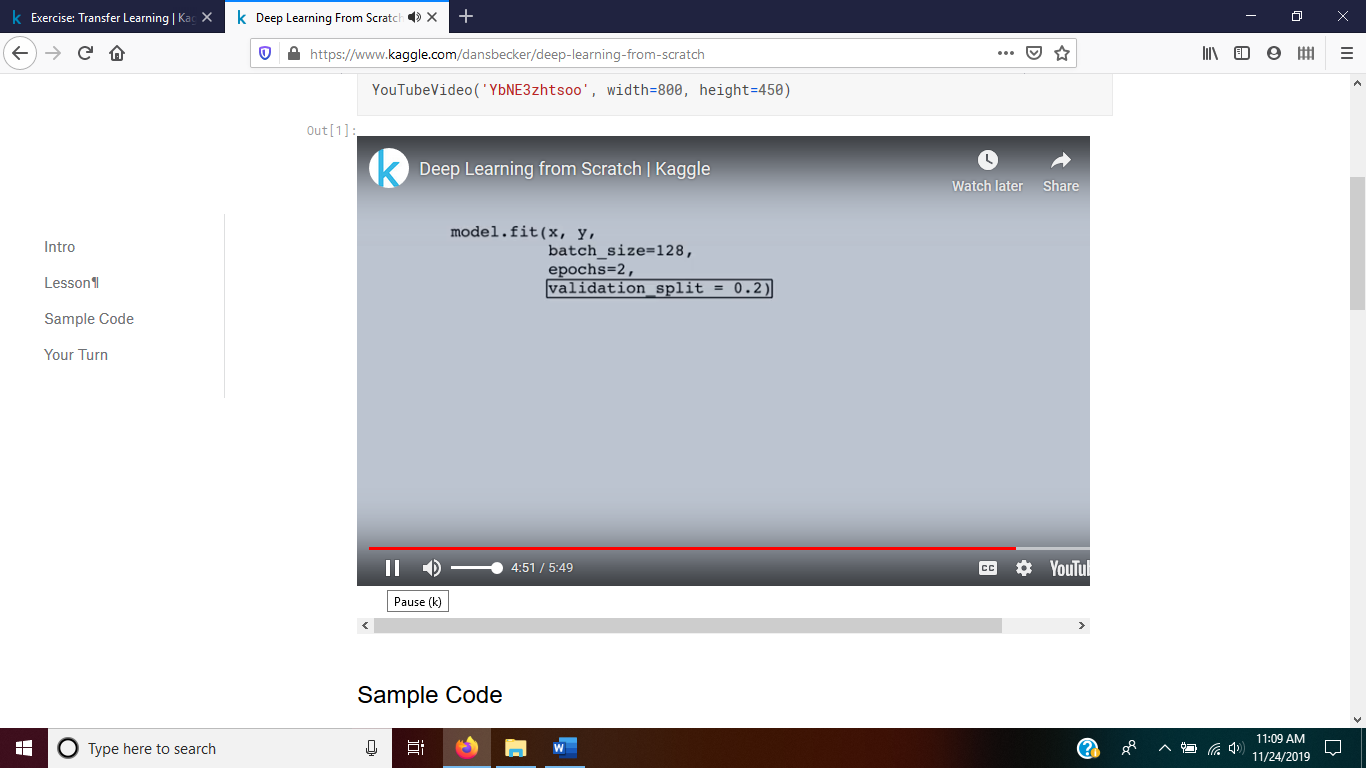
The output layer with the same softmax() probabilities for activation



This compile layer will set the optimization to the ‘adam’ and display ‘accuracy’ instead of loss type metrics



The fit command is used instead of fit\_generator command because the data is loaded into arrays already



The model fits on the training data x and the target data of y, with the set batch size, number of iterations to run the model and the validation of testing data in each iteration (epoch) set aside to test the model set to 20% with 80% to build the model

# Intro

**This is Lesson 8 in the** [**Deep Learning**](https://www.kaggle.com/learn/deep-learning) **track**

At the end of this lesson, you will understand and know how to use

* **Stride lengths** to make your model faster and reduce memory consumption
* **Dropout** to combat overfitting

Both of these techniques are especially useful in large models.

# Lesson

from IPython.display import YouTubeVideo

YouTubeVideo('fwNLf4t7MR8', width=800, height=450)

# Sample Code

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from tensorflow.python import keras

from tensorflow.python.keras.models import Sequential

from tensorflow.python.keras.layers import Dense, Flatten, Conv2D, Dropout

img\_rows, img\_cols = 28, 28

num\_classes = 10

def data\_prep(raw):

out\_y = keras.utils.to\_categorical(raw.label, num\_classes)

num\_images = raw.shape[0]

x\_as\_array = raw.values[:,1:]

x\_shaped\_array = x\_as\_array.reshape(num\_images, img\_rows, img\_cols, 1)

out\_x = x\_shaped\_array / 255

return out\_x, out\_y

train\_size = 30000

train\_file = "../input/digit-recognizer/train.csv"

raw\_data = pd.read\_csv(train\_file)

x, y = data\_prep(raw\_data)

model = Sequential()

model.add(Conv2D(30, kernel\_size=(3, 3),

strides=2,

activation='relu',

input\_shape=(img\_rows, img\_cols, 1)))

model.add(Dropout(0.5))

model.add(Conv2D(30, kernel\_size=(3, 3), strides=2, activation='relu'))

model.add(Dropout(0.5))

model.add(Flatten())

model.add(Dense(128, activation='relu'))

model.add(Dense(num\_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical\_crossentropy,

optimizer='adam',

metrics=['accuracy'])

model.fit(x, y,

batch\_size=128,

epochs=2,

validation\_split = 0.2)

Train on 33600 samples, validate on 8400 samples

Epoch 1/2

33600/33600 [==============================] - 9s 264us/step - loss: 0.6178 - acc: 0.8079 - val\_loss: 0.2279 - val\_acc: 0.9335

Epoch 2/2

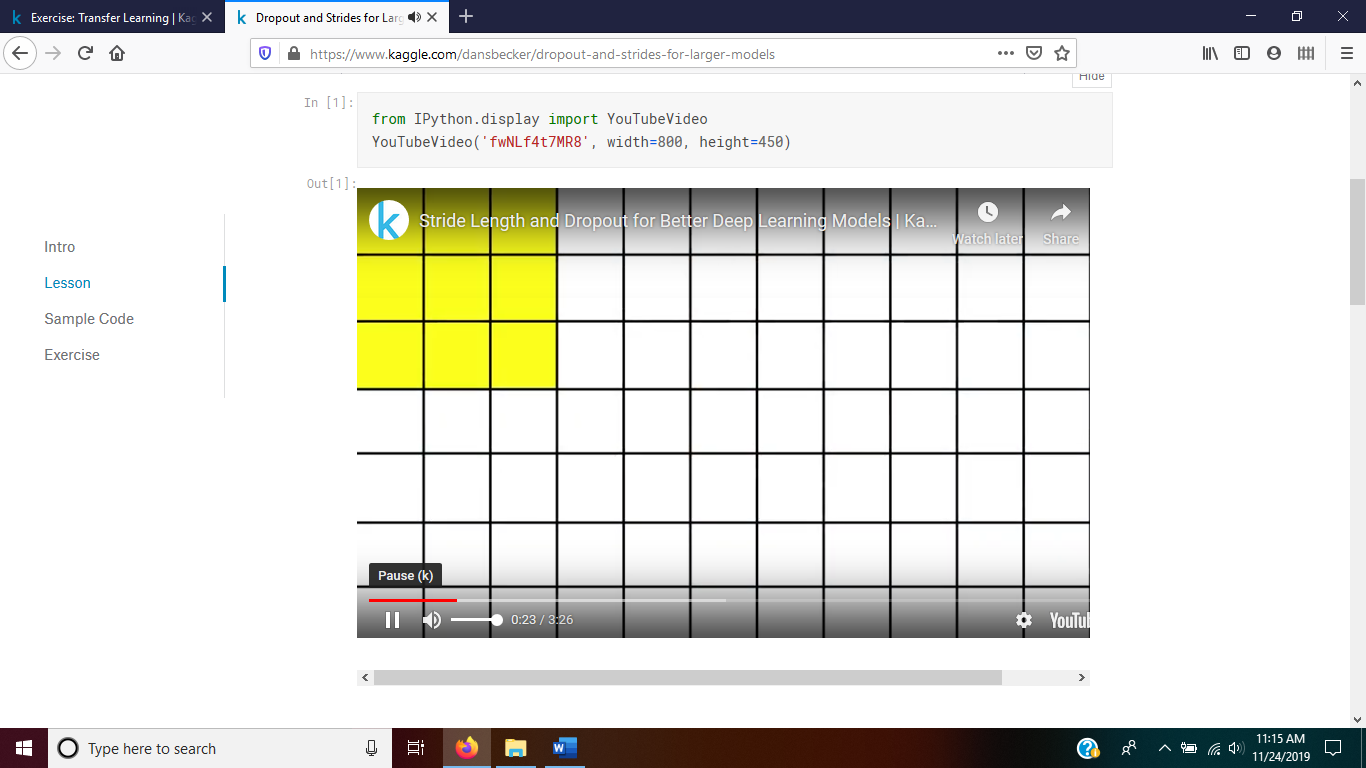
33600/33600 [==============================] - 8s 252us/step - loss: 0.2624 - acc: 0.9212 - val\_loss: 0.1308 - val\_acc: 0.9599

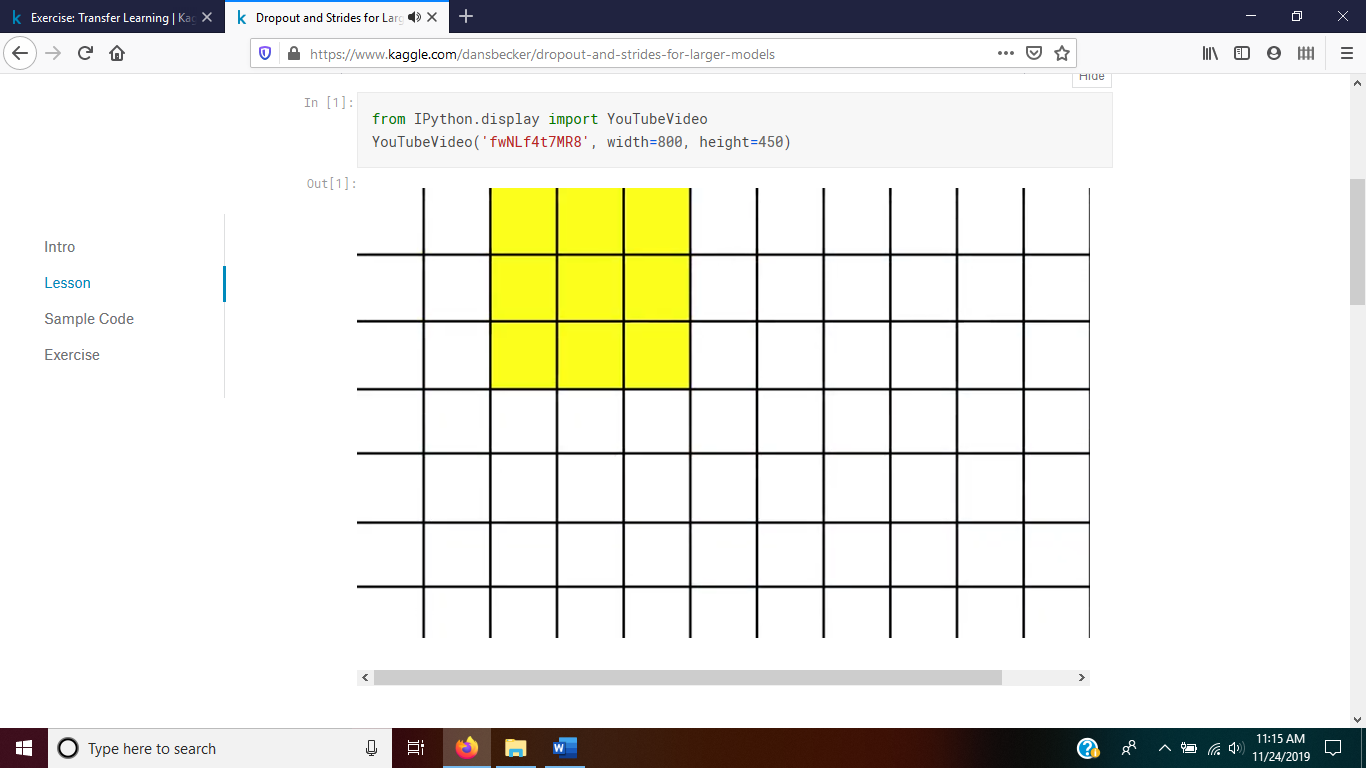
<tensorflow.python.keras.callbacks.History at 0x7f46a26b8828>

# Exercise

[**Apply dropout and strides**](https://www.kaggle.com/kernels/fork/663261) yourself while experimenting with larger models.

[**Deep Learning Course Home Page**](https://www.kaggle.com/learn/deep-learning)



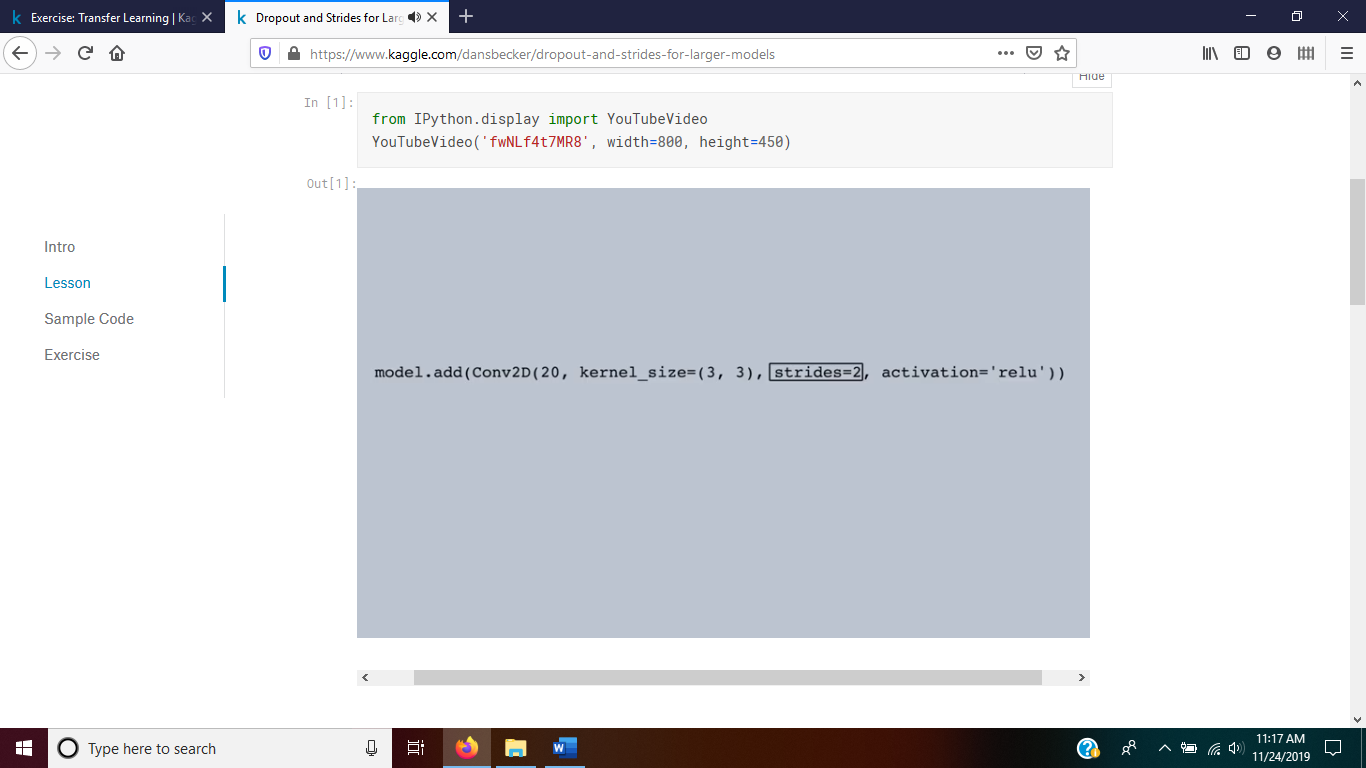


Sliding over 1 or more columns or rows at a time is **striding**

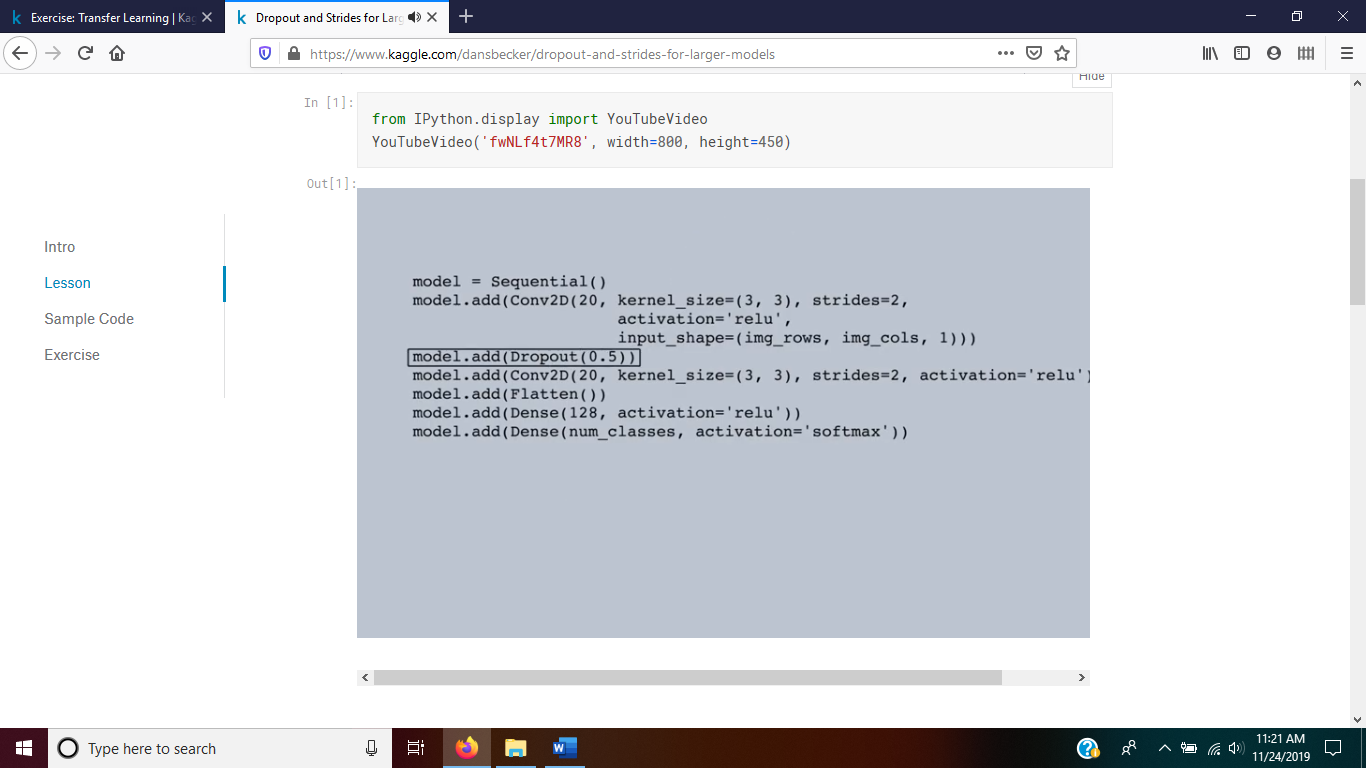
Moving over 2 across makes the image output ½ as wide and 2 down makes output ½ tall

This is in pixels, moving 2 pixels has a stride length of 2

This makes the model much faster, ¼ the size if ½ tall and ½ wide



Another option is max\_pooling that makes the computation faster, but stride setting is cleaner and no systematic difference in model performance between the two

**Dropout** is randomly selecting nodes to use in one iteration or calculation and then randomly choosing a different set of nodes, this is great for preventing overfitting. It allows for each convolution or node find useful information for predictions in its own set without allowing one dominant node or convolution be used repeatedly only tweaking it with other nodes. The above adds a dropout layer and sets the dropout to 50% of the nodes to be dropped in each iteration or ignored/disconnected. These two of striding and dropout our effective in prediction on larger networks for tuning and generalizing the model to not overfit and be more accurate on the testing or validation sets.