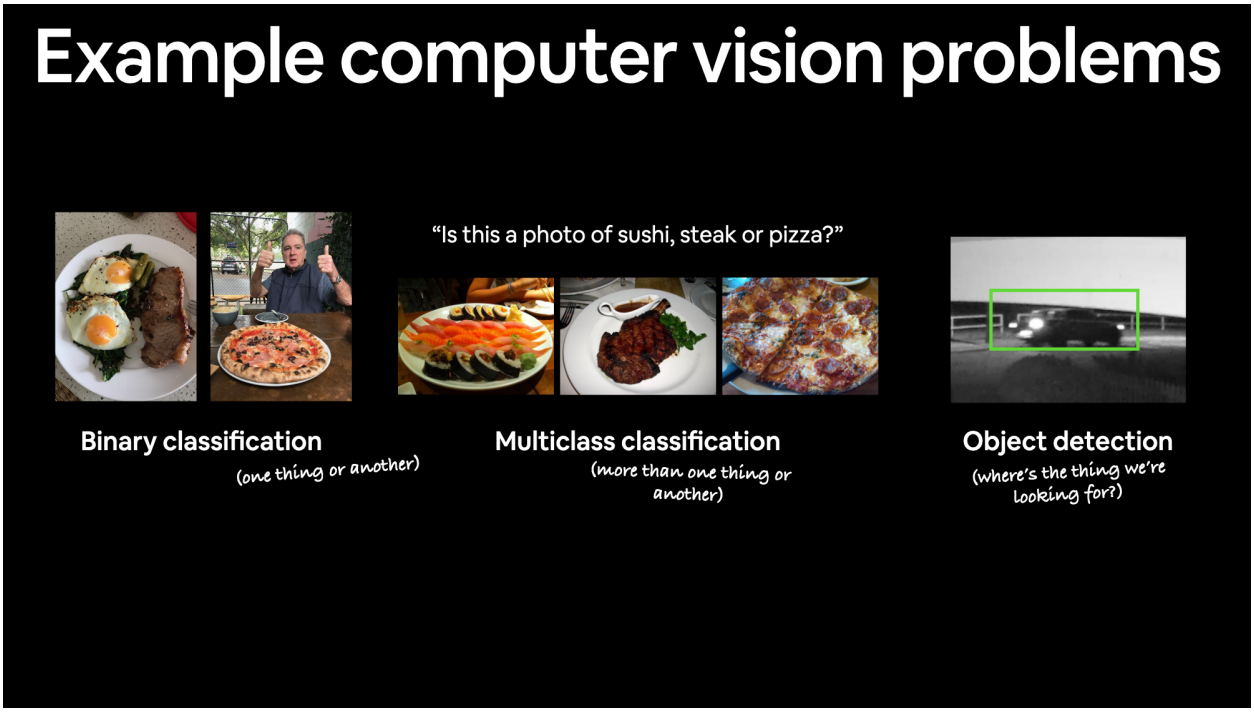
Computer Vision and CNN in TensorFlow

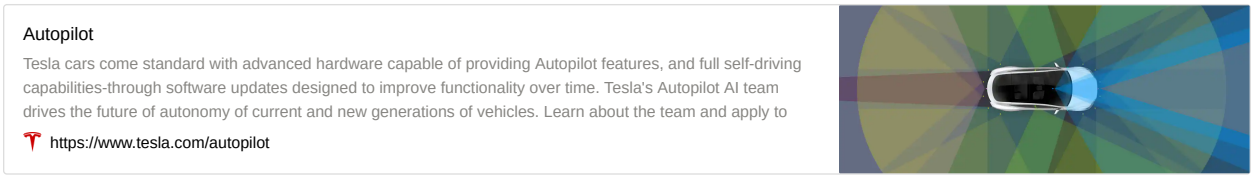
Computer Vision

We want to have a computer to figure out something or understand what’s going on a visual scene.

What is a computer vision problem ?

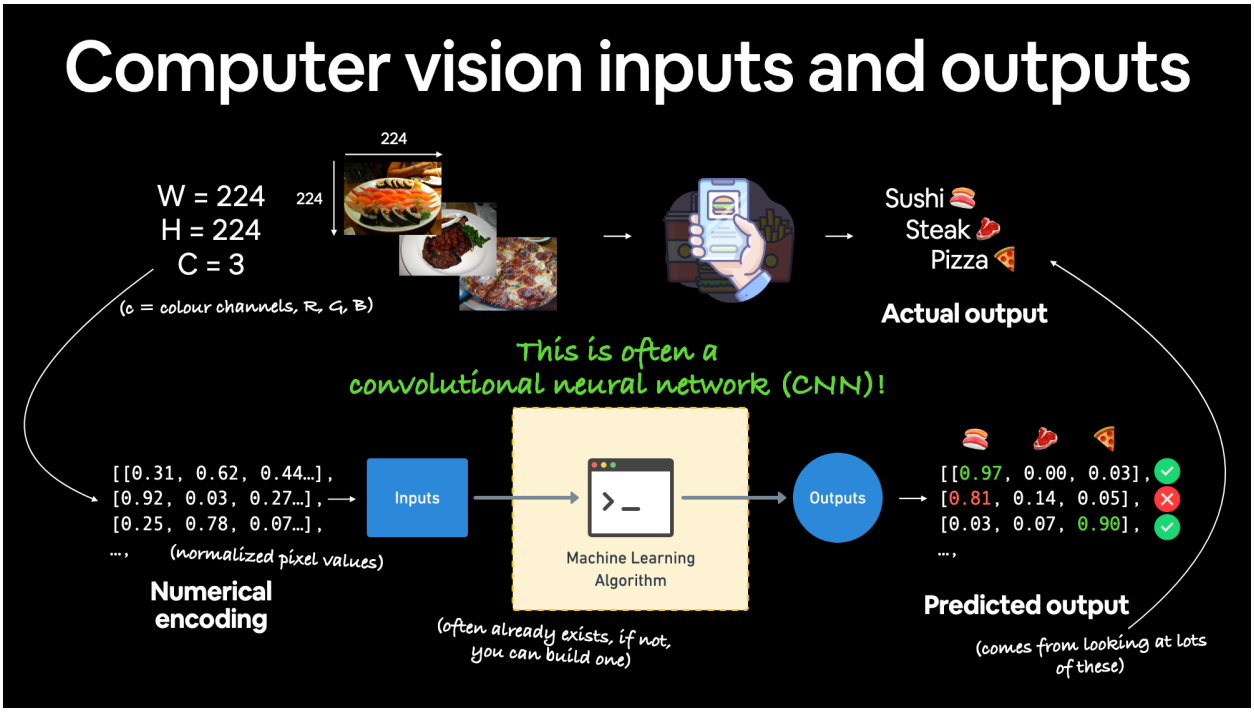


Example → Tesla Autopilot:



A computer vision can be used for almost everything we can image from a visual aspect.

Computer Vision: Inputs & Outputs



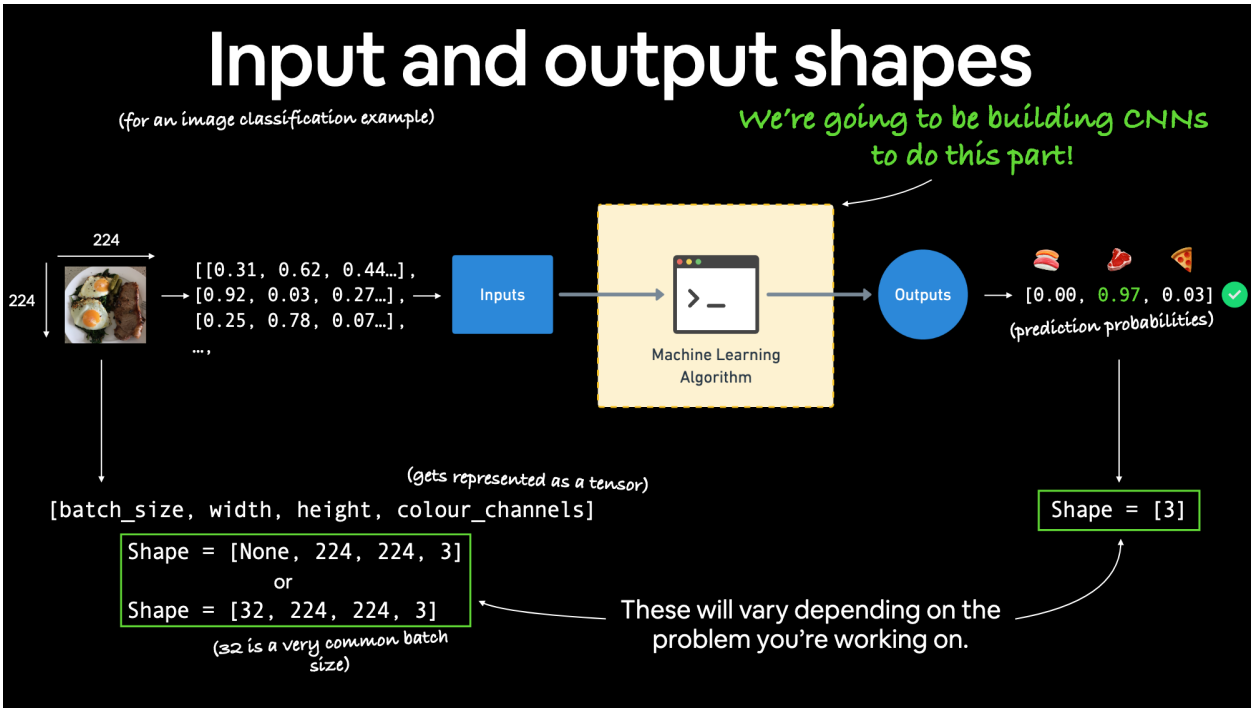
We have images of the food and we want to build an application to identify what's in those photos. With our machine learning algorithm, we might have some inputs and then some outputs.

In our case, our inputs will be numerical encoded versions of those images, and then outputs - prediction probabilities of each class.

Our model would learns this by looking at different examples of actual outputs. Actual examples of images and their proper labels.

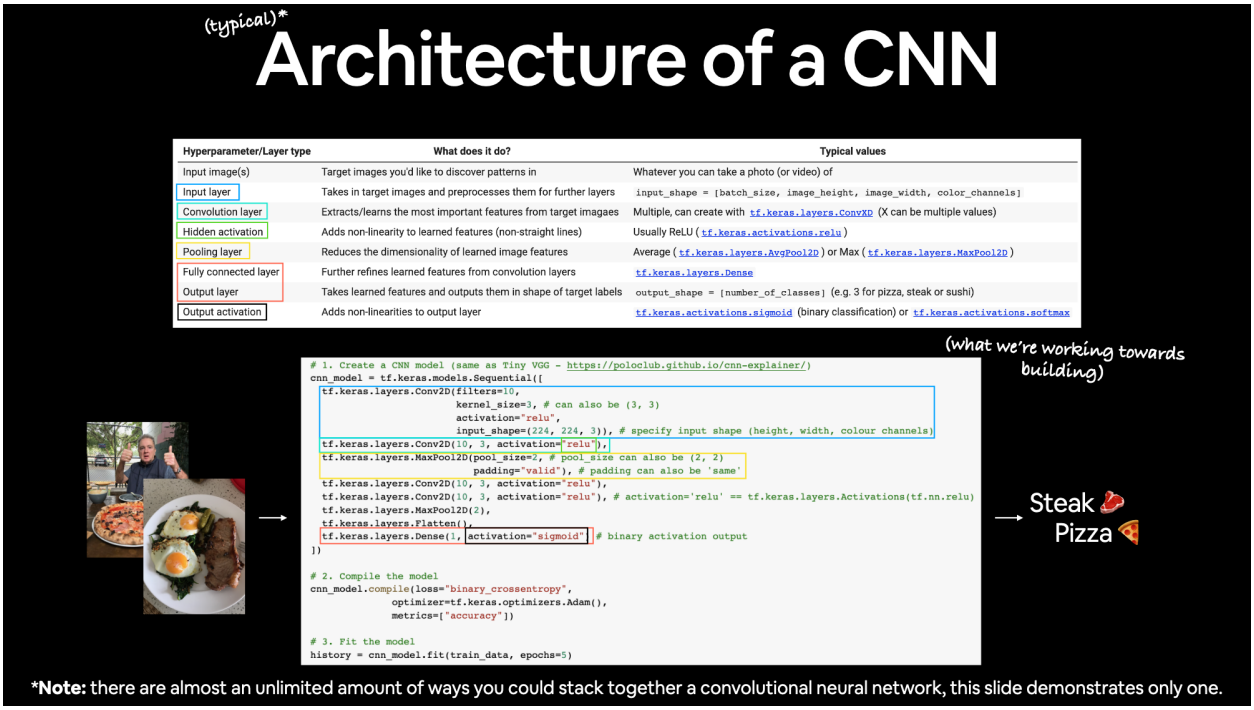
Our machine learning algorithm - is usual Convolutional Neural Network (CNN)

Input & Output Shapes

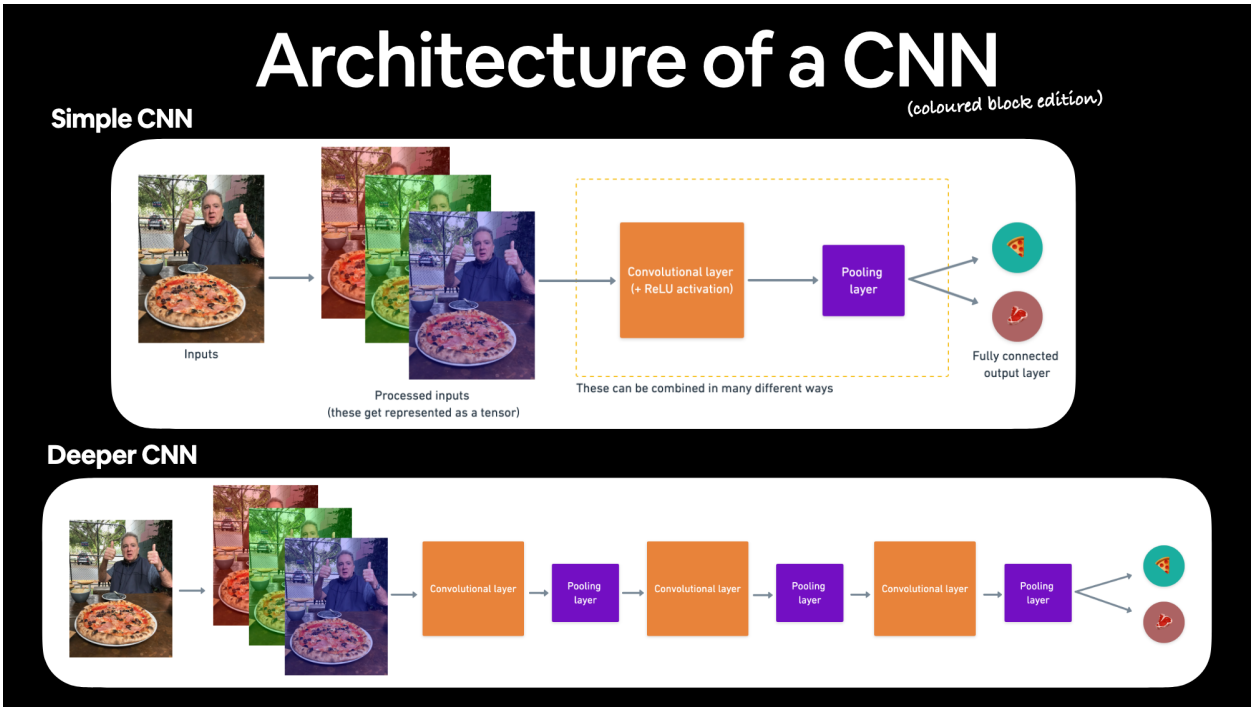


What is a Convolutional Neural Network CNN ?

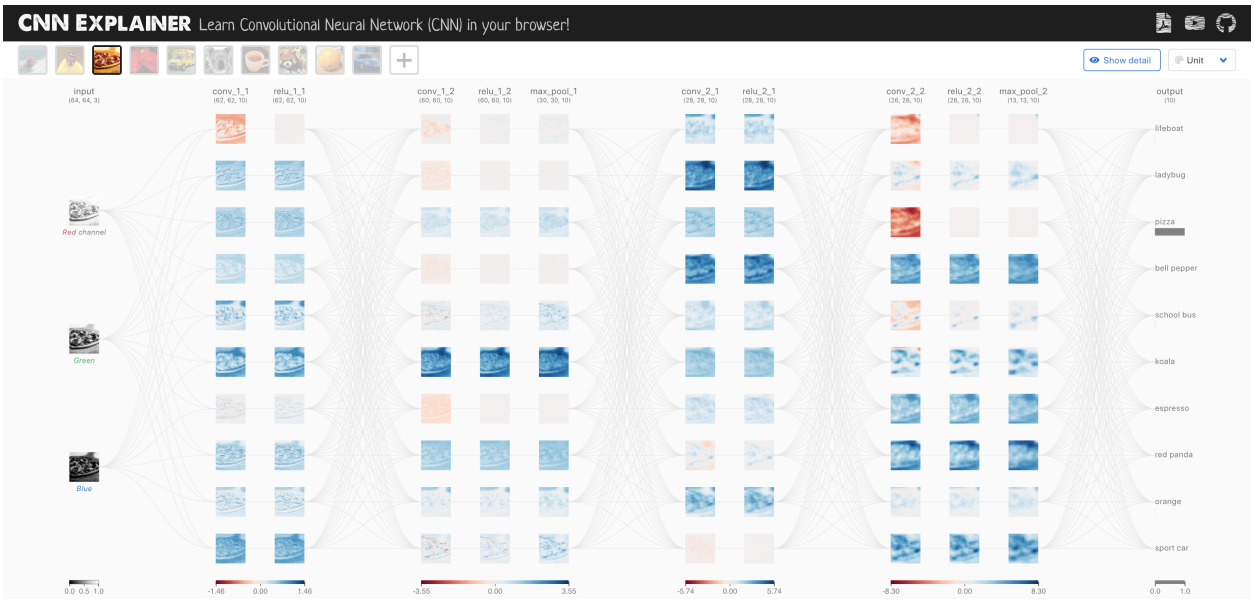
Architecture of a CNN



Hyperparameter/Layer type	What does it do?	Typical values
Input image(s)	Target images we'd like to discover patterns in	Photo or video
Input layer	Takes in target images and preprocess them for further layers	`input_shape = [batch_size, image_height, image_width, color_channels]`
Convolutional layer	Extracts/learns the most important features from target images	Multiple, can create with `tf.keras.layers.ConvXD` (X can be multiple values). There is Conv1D for text data, and Conv3D!
Hidden activation	Adds non-linearity to learned features (non-straight lines)	Usually RELU (`tf.keras.activations.relu`)
Pooling layer	Reduces the dimensionality of learned image features	Average (`tf.keras.layers.AvgPool2D`) or Max (`tf.keras.layers.MaxPool2D`)
Fully connected layer	Further refines learned features from convolution layers	`tf.keras.layers.Dense`
Output layer	Takes learned features and outputs them in a shape of target labels	`output_shape = [number_of_classes]` (e.g. 3 for pizza, steak or sushi)
Output activation	Adds non-linearity to output layer	`tf.keras.activations.sigmoid` (binary classification) or `tf.keras.activations.softmax`) (multi-class classification



CNN Explainer



CNN Explainer

An interactive visualization system designed to help non-experts learn about Convolutional Neural Networks (CNNs).

https://poloclub.github.io/cnn-explainer/

What is a Convolutional Neural Network?

In machine learning, a classifier assigns a class label to a data point. For example, an *image classifier* produces a class label (e.g, bird, plane) for what objects exist within an image. A *convolutional neural network*, or CNN for short, is a type of classifier, which excels at solving this problem!

A CNN is a neural network: an algorithm used to recognize patterns in data. Neural Networks in general are composed of a collection of neurons that are organized in layers, each with their own learnable weights and biases.

1. A **tensor** can be thought of as an n-dimensional matrix. In the CNN above, tensors will be 3-dimensional with the exception of the output layer.
2. A **neuron** can be thought of as a function that takes in multiple inputs and yields a single output. The outputs of neurons are represented above as the red → blue **activation maps**.
3. A **layer** is simply a collection of neurons with the same operation, including the same hyperparameters.
4. **Kernel weights and biases**, while unique to each neuron, are tuned during the training phase, and allow the classifier to adapt to the problem and dataset provided. They are encoded in the visualization with a yellow → green diverging colorscale. The specific values can be viewed in the *Interactive Formula View* by clicking a neuron or by hovering over the kernel/bias in the *Convolutional Elastic Explanation View*.
5. A CNN conveys a **differentiable score function**, which is represented as **class scores** in the visualization on the output layer.

If you have studied neural networks before, these terms may sound familiar to you. So what makes a CNN different? CNNs utilize a special type of layer, aptly named a convolutional layer, that makes them well-positioned to learn from image and image-like data. Regarding image data, CNNs can be used for many different computer vision tasks, such as image processing, classification, segmentation, and object detection.

In CNN Explainer, you can see how a simple CNN can be used for image classification. Because of the network's simplicity, its performance isn't perfect, but that's okay! The network architecture, Tiny VGG, used in CNN Explainer contains many of the same layers and operations used in state-of-the-art CNNs today, but on a smaller scale. This way, it will be easier to understand getting started.

What does each layer of network do ?

Input layer

The input layer (leftmost layer) represents the input image into the CNN. Because we use RGB images as input, the input layer has three channels, corresponding to the red, green, and blue channels, respectively, which are shown in this layer.

Convolutional Layer

The convolutional layers are the foundation of CNN, as they contain the learned kernels (weights), which extract features that distinguish different images from one another—this is what we want for classification!

There are links between the previous layers and the convolutional layers. Each link represents a unique kernel, which is used for the convolution operation to produce the current convolutional neuron's output or activation map.

The convolutional neuron performs an elementwise dot product with a unique kernel and the output of the previous layer's corresponding neuron. . This will yield as many intermediate results as there are unique kernels. The convolutional neuron is the result of all of the intermediate results summed together with the learned bias.

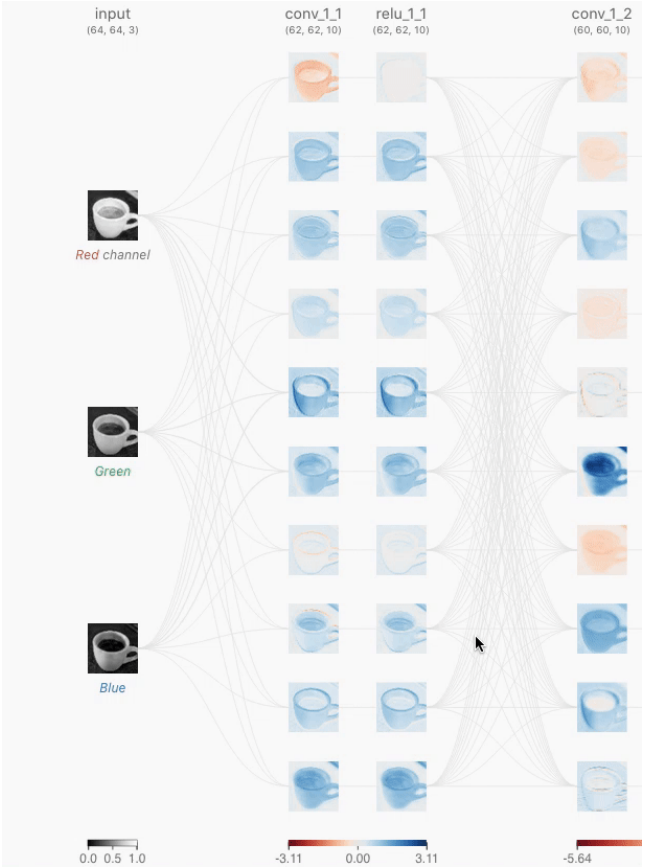
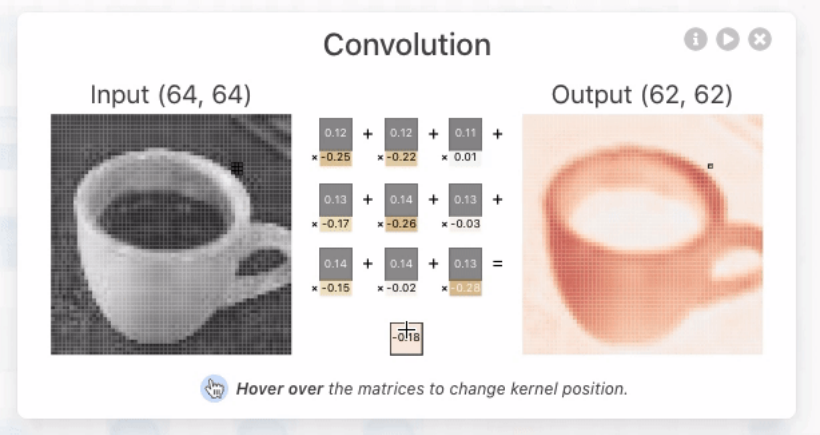


Figure 1. As you hover over the activation map of the topmost node from the first convolutional layer, you can see that 3 kernels were applied to yield this activation map. After clicking this activation map, you can see the convolution operation occuring with each unique kernel.

For example, let's look at the first convolutional layer in the Tiny VGG architecture above. Notice that there are 10 neurons in this layer, but only 3 neurons in the previous layer. In the Tiny VGG architecture, convolutional layers are fully-connected, meaning each neuron is connected to every other neuron in the previous layer. Focusing on the output of the topmost convolutional neuron from the first convolutional layer, we see that there are 3 unique kernels when we hover over the activation map.

The size of these kernels is a hyper-parameter specified by the designers of the network architecture. In order to produce the output of the convolutional neuron (activation map), we must perform an elementwise dot product with the output of the previous layer and the unique kernel learned by the network. In TinyVGG, the dot product operation uses a stride of 1, which means that the kernel is shifted over 1 pixel per dot product, but this is a hyperparameter that the network architecture designer can adjust to better fit their dataset. We must do this for all 3 kernels, which will yield 3 intermediate results.



Then, an elementwise sum is performed containing all 3 intermediate results along with the bias the network has learned. After this, the resulting 2-dimensional tensor will be the activation map viewable on the interface above for the topmost neuron in the first convolutional layer. This same operation must be applied to produce each neuron's activation map.

With some simple math, we are able to deduce that there are 3 x 10 = 30 unique kernels, each of size 3x3, applied in the first convolutional layer. The connectivity between the convolutional layer and the previous layer is a design decision when building a network architecture, which will affect the number of kernels per convolutional layer.

Understanding Hyperparameters

1. **Padding** is often necessary when the kernel extends beyond the activation map. Padding conserves data at the borders of activation maps, which leads to better performance, and it can help preserve the input's spatial size, which allows an architecture designer to build depper, higher performing networks. There exist many padding techniques, but the most commonly used approach is zero-padding because of its performance, simplicity, and computational efficiency. The technique involves adding zeros symmetrically around the edges of an input. This approach is adopted by many high-performing CNNs such as AlexNet.

2. **Kernel size**, often also referred to as filter size, refers to the dimensions of the sliding window over the input. Choosing this hyperparameter has a massive impact on the image classification task. For example, small kernel sizes are able to extract a much larger amount of information containing highly local features from the input. As you can see on the visualization above, a smaller kernel size also leads to a smaller reduction in layer dimensions, which allows for a deeper architecture. Conversely, a large kernel size extracts less information, which leads to a faster reduction in layer dimensions, often leading to worse performance. Large kernels are better suited to extract features that are larger. At the end of the day, choosing an appropriate kernel size will be dependent on your task and dataset, but generally, smaller kernel sizes lead to better performance for the image classification task because an architecture designer is able to stack more and more layers together to learn more and more complex features!

3. **Stride** indicates how many pixels the kernel should be shifted over at a time. For example, as described in the convolutional layer example above, Tiny VGG uses a stride of 1 for its convolutional layers, which means that the dot product is performed on a 3x3 window of the input to yield an output value, then is shifted to the right by one pixel for every subsequent operation. The impact stride has on a CNN is similar to kernel size. As stride is decreased, more features are learned because more data is extracted,

which also leads to larger output layers. On the contrary, as stride is increased, this leads to more limited feature extraction and smaller output layer dimensions. One responsibility of the architecture designer is to ensure that the kernel slides across the input symmetrically when implementing a CNN.

Activation Functions

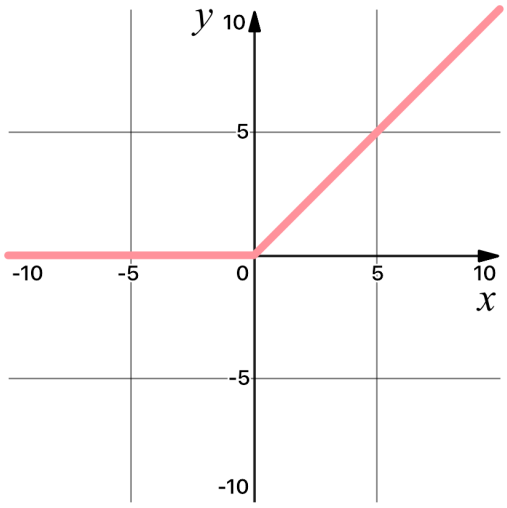
ReLU

Neural networks are extremely prevalent in modern technology—because they are so accurate! The highest performing CNNs today consist of an absurd amount of layers, which are able to learn more and more features. Part of the reason these groundbreaking CNNs are able to achieve such tremendous accuracies is because of their non-linearity.

ReLU applies much-needed non-linearity into the model. Non-linearity is necessary to produce non-linear decision boundaries, so that the output cannot be written as a linear combination of the inputs. If a non-linear activation function was not present, deep CNN architectures would devolve into a single, equivalent convolutional layer, which would not perform nearly as well.

The ReLU activation function is specifically used as a non-linear activation function, as opposed to other non-linear functions such as *Sigmoid* because it has been empirically observed that CNNs using ReLU are faster to train than their counterparts.

The ReLU activation function is a one-to-one mathematical operation:



This activation function is applied elementwise on every value from the input tensor. For example, if applied ReLU on the value 2.24, the result would be 2.24, since 2.24 is larger than 0.

The Rectified Linear Activation function (ReLU) is performed after every convolutional layer in the network architecture outlined above.

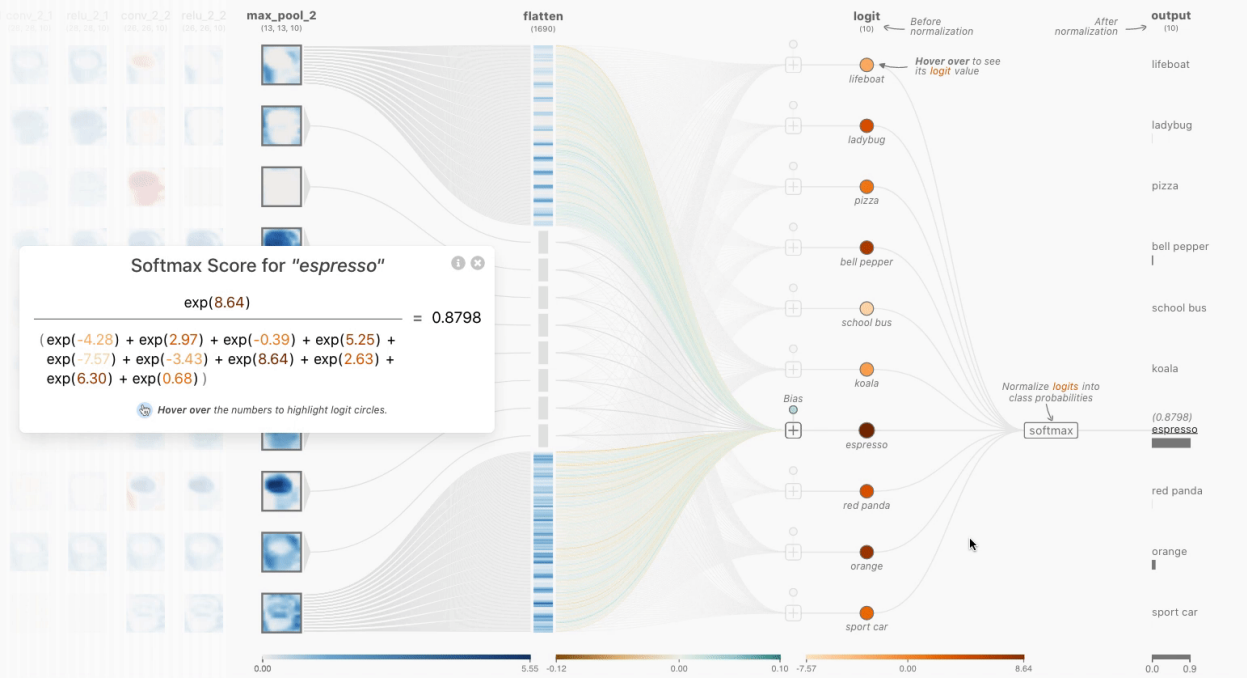
SoftMax

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

A softmax operation serves a key purpose: making sure the CNN outputs sum to 1. Because of this, softmax operations are useful to scale model outputs into probabilities.

Notice how the logits after flatten aren't scaled between zero to one. For a visual indication of the impact of each logit (unscaled scalar value), they are encoded using a light orange → dark orange color scale. After passing through the softmax function, each class now corresponds to an appropriate probability!

You might be thinking what the difference between standard normalization and softmax is—after all, both rescale the logits between 0 and 1. Remember that backpropagation is a key aspect of training neural networks—we want the correct answer to have the largest "signal." By using softmax, we are effectively "approximating" argmax while gaining differentiability. Rescaling doesn't weigh the max significantly higher than other logits, whereas softmax does. Simply put, softmax is a "softer" argmax



Pooling Layers

There are many types of pooling layers in different CNN architectures, but they all have the purpose of gradually decreasing the spatial extent of the network, which reduces the parameters and overall computation of the network. The type of pooling used in the Tiny VGG architecture above is Max-Pooling.

The Max-Pooling operation requires selecting a kernel size and a stride length during architecture design. Once selected, the operation slides the kernel with the specified stride over the input while only selecting the largest value at each kernel slice from the input to yield a value for the output.

In the Tiny VGG architecture above, the pooling layers use a 2x2 kernel and a stride of 2. This operation with these specifications results in the discarding of 75% of activations. By discarding so many values, Tiny VGG is more computationally efficient and avoids overfitting.

Flatten Layer

This layer converts a three-dimensional layer in the network into a one-dimensional vector to fit the input of a fully-connected layer for classification. For example, a 5x5x2 tensor would be converted into a vector of size 50. The previous convolutional layers of the network extracted the features from the input image, but now it is time to classify the features. We use the softmax function to classify these features, which requires a 1-dimensional input. This is why the flatten layer is necessary.

Becoming one with the Data

```
# Get the classnames programmatically
import pathlib
import numpy as np
data_dir = pathlib.Path("pizza_steak/train")
class_names = np.array(sorted([item.name for item in data_dir.glob("*")])) # Created a list of class_names from the subdirectories
print(class_names)
```

```
# Let's visualize our images
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import random

def view_random_image(target_dir, target_class):
  # Setup the target directory (we will view the images from here)
  target_folder = target_dir + target_class

  # Get a random image path
  random_image = random.sample(os.listdir(target_folder), 1) # Randomly sample 1 of the items specified in target folder

  # Read the image and plot using matplotlib
  img = mpimg.imread(target_folder + "/" + random_image[0])
  plt.imshow(img)
  plt.title(target_class)
  plt.axis("off")

  print(f"Image shape: {img.shape}") # Show the shape of the image

  return img
```

🤔 **Note**
Many Machine Learning models, including neural networks prefer values they work with to be 0 and 1. Knowing this, one of the most common preprocessing steps for working with images is to **scale** (also referred as **normalize**) their pixel values by dividing the image arrays by 255 (since 255 is the maximum pixel value)

End-to-End Example

- Load our images
- Preprocess our images
- Build a CNN to find patterns in our images
- Compile our CNN
- Fit CNN to our training data

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Set the seed
tf.random.set_seed(42)

# Preprocess data (get all of the pixel values between 0 and 1, also called scaling/normalization)
train_datagen = ImageDataGenerator(rescale=1./255)
valid_datagen = ImageDataGenerator(rescale=1./255)

# Setup paths to our data directories
train_dir = "pizza_steak/train"
test_dir = "pizza_steak/test"

# Import data from directories and turn it into batches -- Read Documentation
# It creates data and label automatically for us
train_data = train_datagen.flow_from_directory(directory=train_dir,
                                               batch_size=32,
                                               target_size=(244,244),# 244,244 common shape
                                               class_mode="binary",
                                               seed=42)
valid_data = valid_datagen.flow_from_directory(directory=test_dir,
                                               batch_size=32,
                                               target_size=(244,244),
                                               class_mode="binary",
                                               seed=42)

# Build a CNN model (same as the Tiny VGG on the CNN explainer website)
model_1 = tf.keras.models.Sequential([
      tf.keras.layers.Conv2D(filters=10,
                             kernel_size=3,
                             activation="relu",
                             input_shape=(244,244,3)),
      tf.keras.layers.Conv2D(10,3,activation="relu"),
      tf.keras.layers.MaxPool2D(pool_size=2,
                                padding="valid"),
      tf.keras.layers.Conv2D(10,3,activation="relu"),
      tf.keras.layers.Conv2D(10,3),
      tf.keras.layers.Activation(tf.nn.relu), # We can setup the activation function like this, this will be the same as specifying the arg
      tf.keras.layers.MaxPool2D(pool_size=2),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(1, activation="sigmoid")
])

# Compile our CNN
model_1.compile(loss="binary_crossentropy",
                optimizer=tf.keras.optimizers.Adam(),
                metrics=["accuracy"])

# Fit the model
history_1 = model_1.fit(train_data, epochs=5,
                        steps_per_epoch=len(train_data),
                        validation_data=valid_data,
                        validation_steps=len(valid_data))
```

Model summary

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_8 (Conv2D)           (None, 242, 242, 10)      280

 conv2d_9 (Conv2D)           (None, 240, 240, 10)      910

 max_pooling2d_3 (MaxPooling  (None, 120, 120, 10)     0
 2D)

 conv2d_10 (Conv2D)          (None, 118, 118, 10)      910

 conv2d_11 (Conv2D)          (None, 116, 116, 10)      910

 activation_1 (Activation)   (None, 116, 116, 10)      0

 max_pooling2d_4 (MaxPooling  (None, 58, 58, 10)       0
 2D)

 flatten_1 (Flatten)         (None, 33640)             0

 dense_1 (Dense)             (None, 1)                 33641

=================================================================
Total params: 36,651
Trainable params: 36,651
Non-trainable params: 0
_________________________________________________________________
```

🤔 **Note**

You can think of trainable parameters as patterns a model can learn from data. Intuitively, you might think more is better. And in lots of cases, it is. But in this case, the difference here is the two different styles of model we're using. Where a series of dense layers has a number of different learnable parameters connected to each other and hence a higher number of possible learnable patterns, a convolutional neural network seeks to sort out and learn the most important patterns in an image. So even though these are less learnable parameters in our convolutional neural network, these are often more helpful in dechiphering between different features in an image.

❓ Whenever we see a ValueError that something about shape, most likely we have some sort of shape mismatch in the model architecture. In Deep Learning model, each layer output is subsequent layers input.

Binary Classification Example

1. Become one with the data
2. Preprocess the data
3. Create a model (Baseline model)
4. Fit the model
5. Evaluate the model
6. Adjust different parameters and improve the model
7. Repeat until satisfied

Become one with the data

```
# Visualize data
plt.figure()
plt.subplot(1,2,1)
steak_img = view_random_image("pizza_steak/train/", "steak")
plt.subplot(1,2,2)
pizza_img = view_random_image("pizza_steak/train/", "pizza")
```

Preprocess the data

```
# Define directory dataset paths
train_dir = "pizza_steak/train/"
test_dir = "pizza_steak/test/"
```

Turn our data into **batches**.

A **batch** - is a small subset of our dataset that the model looks during the training. Rather than look at all ~10k images at one time, a model might only look at 32 at a time.

It does this for couple of reasons:

1. 10,000 (or more) images might not fit into the memory of processor (GPU)
2. Trying to learn the patterns in 10,000 images in one hit could result in the model not being able to learn very well.

```
# Create train and test data generators and rescale the data
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Normalize (rescale) the images
train_datagen = ImageDataGenerator(rescale=1/255.)
test_datagen = ImageDataGenerator(rescale=1/255.)

# Load in our image data from directories and turn them into batches
train_data = train_datagen.flow_from_directory(directory=train_dir, # Target directory of images
                                               target_size=(244,244), # Target size of images (height, width)
                                               class_mode="binary", # Type of data we're working with
                                               batch_size=32 # size of minibatches to load data into
                                              )

test_data = test_datagen.flow_from_directory(directory=test_dir,
                                             target_size=(244,244),
                                             class_mode="binary",
                                             batch_size=32)


# Get a sample of a train data batch
images, labels = train_data.next() # get the next batch of images/labels in train data
len(images), len(labels)

# How many batches are there
len(train_data)

# View the first batch of labels
labels
```

Create a CNN model (Baseline model)

A baseline is a relatively simple model or existing result that you setup when beginning a machine learning experiment and then as you keep experimenting, you try to beat the baseline.

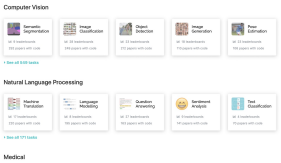
🤔 **Note**
In Deep Learning, there is almost an infinite amount of architectures you could create. So one of the best ways to get started is to start with something simple and see if it works on your data and then introduce complexity as required (e.g. look at which current model is performing best in the field for your problem

Resource to look for best performing models:

Papers with Code - Browse the State-of-the-Art in Machine Learning

7422 leaderboards * 3063 tasks * 6151 datasets * 70109 papers with code.

https://paperswithcode.com/sota




```
# Make the creating of our model a little easier
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPool2D, Activation
from tensorflow.keras import Sequential

# Create the model
model_4 = Sequential([
      Conv2D(filters=10, # filters is the number of sliding windows going across an input (higher = more complex model)
             kernel_size=3, # the size of the sliding window going across an input
             strides=1, # the size of the step the sliding window takes across an input
             padding="valid", # if "same", output shape is same as input shape, if "valid" output shape gets compressed
             activation="relu",
             input_shape=(244,244,3)), # Input Layer (specify the input shape)
      Conv2D(10, 3, activation="relu"),
      Conv2D(10, 3, activation="relu"),
      Flatten(),
      Dense(1, activation="sigmoid") # Output Layer
])
```

Breakdown of Conv2D Layer

Breakdown of `Conv2D` layer

Example code: `tf.keras.layers.Conv2D(filters=10, kernel_size=(3, 3), strides=(1, 1), padding="same")`
Example 2 (same as above): `tf.keras.layers.Conv2D(filters=10, kernel_size=3, strides=1, padding="same")`

Hyperparameter name	What does it do?	Typical values
Filters	Decides how many filters should pass over an input tensor (e.g. sliding windows over an image).	`10, 32, 64, 128` (higher values lead to more complex models)
Kernel size (also called filter size)	Determines the shape of the filters (sliding windows) over the output.	`3, 5, 7` (lowers values learn smaller features, higher values learn larger features)
Padding	Pads the target tensor with zeroes (if "`same`") to preserve input shape. Or leaves in the target tensor as is (if "`valid`"), lowering output shape.	"same" or "valid"
Strides	The number of steps a filter takes across an image at a time (e.g. if `strides=1`, a filter moves across an image 1 pixel at a time).	`1` (default), `2`

📖 **Resource:** For an interactive demonstration of the above hyperparameters, see the CNN explainer website.

Fit the model

```
# Fit the model
history_4 = model_4.fit(train_data, # Combination of labels and sample data
                        epochs=5,
```

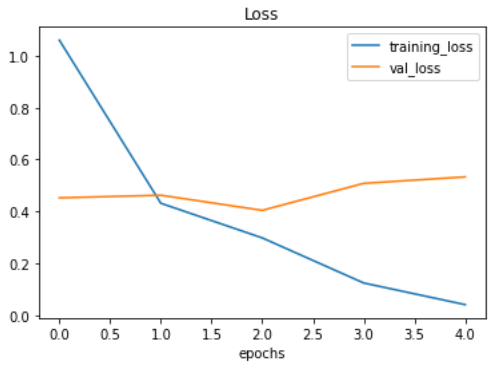
```
                    steps_per_epoch=len(train_data),
                    validation_data=test_data,
                    validation_steps=len(test_data))
```

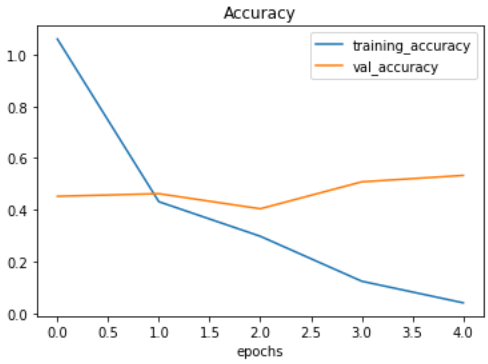
Evaluating the model

```
# Plot the validation and training curves separately
def plot_loss_curves(history):
  """
  Returns separate loss curves for training and validation metrics
  """

  loss = history.history["loss"]
  val_loss = history.history["val_loss"]
  accuracy = history.history["accuracy"]
  val_accuracy = history.history["val_accuracy"]

  epochs = range(len(history.history["loss"]))

  # Plot loss
  plt.plot(epochs, loss, label="training_loss")
  plt.plot(epochs, val_loss, label="val_loss")
  plt.title("Loss")
  plt.xlabel("epochs")
  plt.legend()

  # Plot accuracy
  plt.figure()
  plt.plot(epochs, loss, label="training_accuracy")
  plt.plot(epochs, val_loss, label="val_accuracy")
  plt.title("Accuracy")
  plt.xlabel("epochs")
  plt.legend();
```

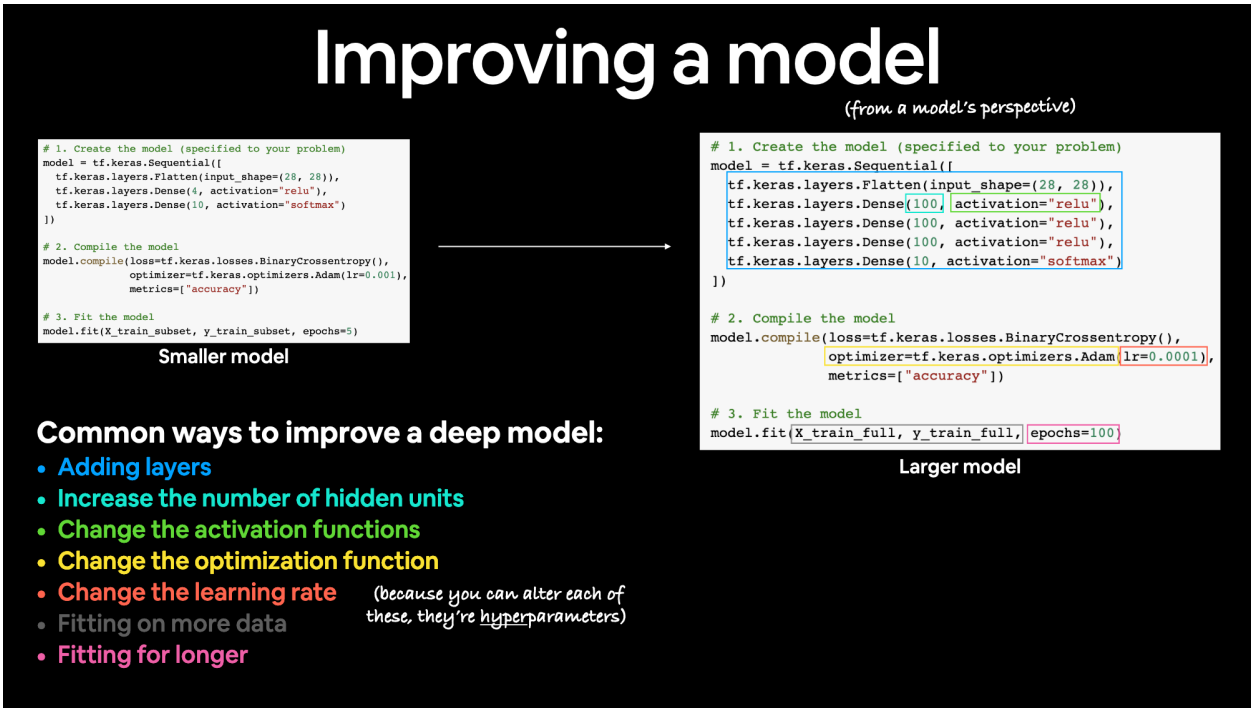
🤔 **Note**

When a validation loss starts to increase, it's likely that the model is *overfitting* the training dataset. This means it's learning the patterns in the training dataset *too well* and thus the model's ability to generalize to unseen data will be dimished.





Adjust Model Parameters



Fitting a Machine Learning model comes in 3 steps:

1. Create a baseline model
2. Beat the baseline model by overfitting a larger model
3. Reduce overfitting

Ways to induce overfitting:

- Increase the number of conv layers
- Increase the number of filters
- Add another Dense layer to the output of our flattened layer

Reduce overfitting:

- Add data augmentation
- Add regularization layers (such as MaxPool2D)
- Add more data

Reducing overfitting is also known as regularization.

```
# Create the model
model_5 = Sequential([
    Conv2D(10, 3, activation="relu", input_shape=(244,244,3)),
    MaxPool2D(pool_size=2),
    Conv2D(10, 3, activation="relu"),
    MaxPool2D(),
    Conv2D(10,3,activation="relu"),
    MaxPool2D(),
    Flatten(),
    Dense(1, activation="sigmoid")
])

# Compile the model
model_5.compile(loss="binary_crossentropy",
                optimizer=Adam(),
                metrics=["accuracy"])

# Fit the model
history_5 = model_5.fit(train_data,
                        epochs=5,
                        steps_per_epoch=len(train_data),
                        validation_data=test_data,
                        validation_steps=len(test_data))

# Get a summary of our model with max pooling
model_5.summary()
```

Data Augmentation

Data augmentation is the process of altering our training data, leading it have more diversity and in turn allowing our models to learn mode generalizable patterns. Altering might mean adjusting the rotation of an image, flipping it, cropping it or something similar.

tf.keras.preprocessing.image.ImageDataGenerator | TensorFlow Core v2.8.0

Generate batches of tensor image data with real-time data augmentation.

https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator

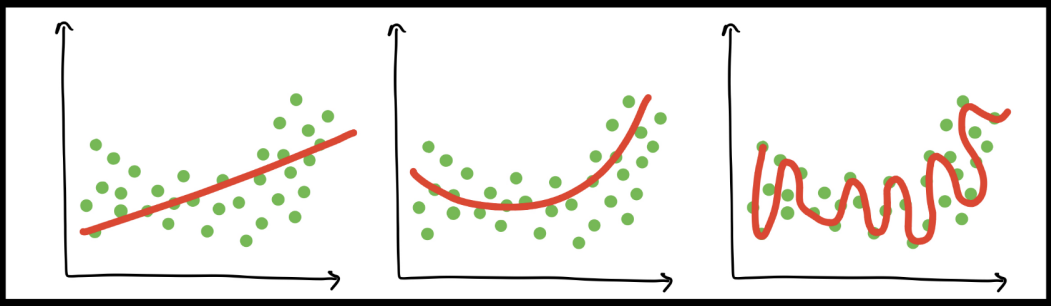
```
# Create ImageDataGenerator training instance with data augmentation
train_datagen_augmented = ImageDataGenerator(rescale=1/255.,
                                             rotation_range=0.2, # how much do you want to rotate an image?
                                             shear_range=0.2, # how much do you shear the image?
                                             zoom_range=0.2, # zoom in randomly on an image
                                             width_shift_range=0.2, # move your image around on the x-axis
                                             height_shift_range=0.3, # move your image around on the y-axis
                                             horizontal_flip=True) # do you want to flip an image?

# Create ImageDataGenerator without data augmentation
train_datagen = ImageDataGenerator(rescale=1/255.)

# Create ImageDataGenerator without data augmentation for the test dataset
test_datagen = ImageDataGenerator(rescale=1/255.)
```

Overfitting

What is overfitting?

Overfitting — when a model over learns patterns in a particular dataset and isn't able to generalise to unseen data.

For example, a student who studies the course materials too hard and then isn't able to perform well on the final exam. Or tries to put their knowledge into practice at the workplace and finds what they learned has nothing to do with the real world.



Underfitting | **Balanced** (goldilocks zone) | **Overfitting**

Improving the Model to reduce overfitting

Improving a model

(from a data perspective)

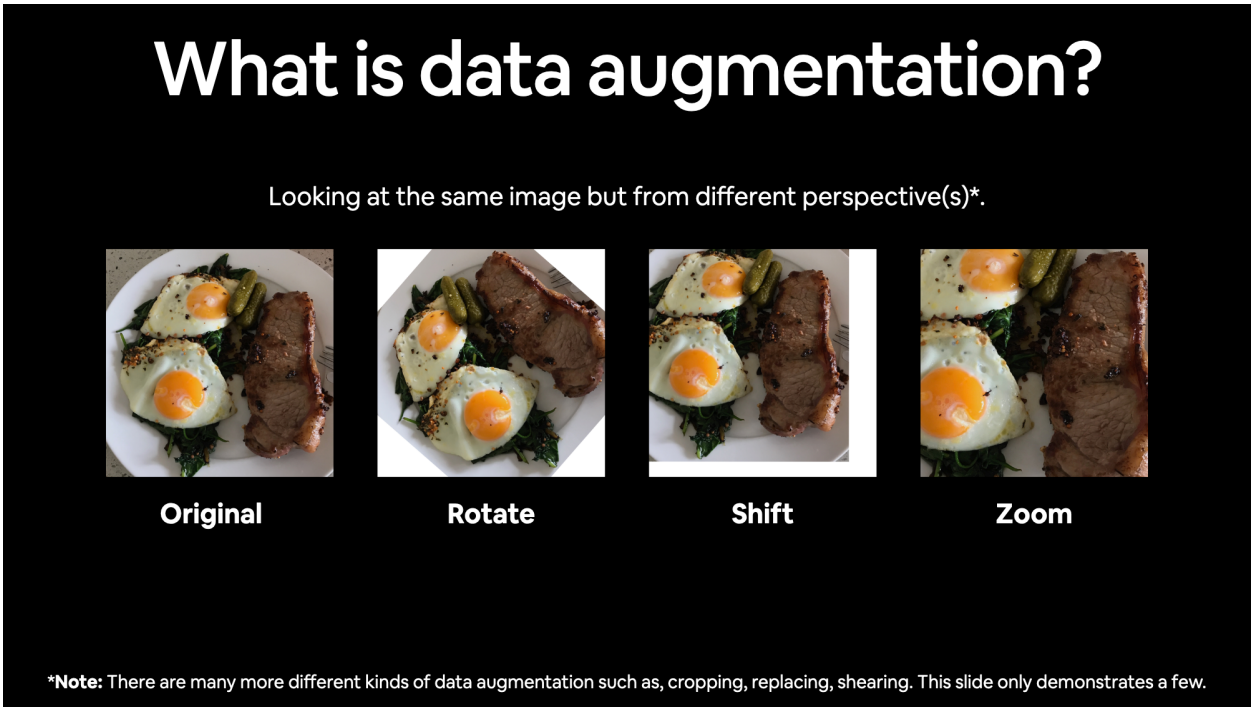
Method to improve a model (reduce overfitting)	What does it do?
More data	Gives a model more of a chance to learn patterns between samples (e.g. if a model is performing poorly on images of pizza, show it more images of pizza).
Data augmentation	Increase the diversity of your training dataset without collecting more data (e.g. take your photos of pizza and randomly rotate them 30°). Increased diversity forces a model to learn more generalisation patterns.
Better data	Not all data samples are created equally. Removing poor samples from or adding better samples to your dataset can improve your model's performance.
Use transfer learning	Take a model's pre-learned patterns from one problem and tweak them to suit your own problem. For example, take a model trained on pictures of cars to recognise pictures of trucks.

Data Augmentation Continued

```
# Import data and augment it from the training directory
print("Augmented Training Data")
train_data_augmented = train_datagen_augmented.flow_from_directory(train_dir,
                                                                   target_size=(244,244),
                                                                   batch_size=32,
                                                                   class_mode="binary",
                                                                   shuffle=False) # For demonstration purposes only

# Create non-augmented train data batches
print("Non-augmented Training Data")
train_data = train_datagen.flow_from_directory(train_dir,
                                               target_size=(244,244),
                                               batch_size=32,
                                               class_mode="binary",
                                               shuffle=False)

# Create non-augmented test data batches
print("Non-augmented Test Data")
test_data = test_datagen.flow_from_directory(test_dir,
                                             target_size=(244,244),
                                             batch_size=32,
                                             class_mode="binary")
```

🤔 **Note**

Data Augmentation is usually only performed on the training data. Using `ImageDataGenerator` built-in data augmentation parameters our images are left as they are in the directories but are modified as they're loaded into them model.




```
# Get sample data batches
images, label = train_data.next()
augmented_images, augmented_labels = train_data_augmented.next() # Labels are not augmented, only the data!

# Show original image and augmented image
import random
```

```
random_number = random.randint(0,32) # bath_size = 32
print(f"Showing image number: {random_number}")
plt.imshow(images[random_number])
plt.title(f"Original Image")
plt.axis(False)

plt.figure()
plt.imshow(augmented_images[random_number])
plt.title(f"Augmneted Image")
plt.axis(False);
```

```
# Create a model (same as model_5)
model_6 = Sequential([
      Conv2D(10, 3, activation="relu"),
      MaxPool2D(pool_size=2),
      Conv2D(10, 3, activation="relu"),
      MaxPool2D(),
      Conv2D(10,3, activation="relu"),
      MaxPool2D(),
      Flatten(),
      Dense(1, activation="sigmoid")
])

# Compile the model
model_6.compile(loss="binary_crossentropy",
                optimizer=Adam(),
                metrics=["accuracy"])

# Fit the model
history_6 = model_6.fit(train_data_augmented,
                        epochs=5,
                        steps_per_epoch=len(train_data_augmented),
                        validation_data=test_data,
                        validation_steps=len(test_data))
```

Shuffle the Dataset

```
# Import data and augment it and shuffle from the training directory
train_data_augmented_shuffled = train_datagen_augmented.flow_from_directory(train_dir,
                                                                            target_size=(244,244),
                                                                            class_mode="binary",
                                                                            batch_size=32,
                                                                            shuffle=True)
```

```
# Create the model (same as model_5 and model_6)
model_7 = Sequential([
      Conv2D(10, 3, activation="relu", input_shape=(244,244,3)),
      MaxPool2D(),
      Conv2D(10, 3, activation="relu"),
      MaxPool2D(),
      Conv2D(10,3, activation="relu"),
      MaxPool2D(),
      Flatten(),
      Dense(1, activation="sigmoid")
])

# Compile the model
model_7.compile(loss="binary_crossentropy",
                optimizer=Adam(),
                metrics=["accuracy"])

# Fit the model
history_7 = model_7.fit(train_data_augmented_shuffled,
                        epochs=5,
                        steps_per_epoch=len(train_data_augmented_shuffled),
```

```
                    validation_data=test_data,
                    validation_steps=len(test_data))
```

🤔 **Note**
When shuffling training data, the model gets exposed to all different kinds of data during training, thus enabling it to learn featrues across a wide array of images.

Ways of Model Improvement

We can improve our model:

1. Increase the number of model layers (e.g add more `Conv2D` / `MaxPool2D` layers)
2. Increase the number of filters in each convolutional layer (e.g from 10 to 32 or even 64)
3. Train for longer (more epochs)
4. Find an idal learning rate
5. Get more data
6. Use **transfer learning** to leverage what another image model has learned and adjust it for our own use case.

Making a Prediction with our trained model on our own custom data

```
# Classes we're working with
print(class_names)
# View our example image
import matplotlib.image as mpimg
!wget https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/images/03-steak.jpeg
steak = mpimg.imread("03-steak.jpeg")
```

🤔 **Note**
When you train a neural network and you want to make a prediction with it on your own custom data, it's important than your own custom data (or new data) is preprocessed into the same format as the data your model was trained on

```
# Create a function to import an image and resize it to be able to be used with our model
def load_and_prep_image(filename, img_shape=244):
  """
  Reads the image from filename, and turns it into a tensor,
  reshapes it (img_shape, img_shape, colour_channels)
  """

  # Read in the image
  img = tf.io.read_file(filename)
  # Decode the read file into a tensor
  img = tf.image.decode_image(img)
  # Resize the image
  img = tf.image.resize(img, size=[img_shape, img_shape])
  # Rescale the image (normalize)
  img = img/255.

  return img


steak = load_and_prep_image("03-steak.jpeg")
steak
```

```
def pred_and_plot(model, filename, class_names=class_names):
  """
  Imports an image located at filename, makes a prediction with model
  and plots the image with the predicted class as the title.
  """
  # Import the target image and preprocess it
  img = load_and_prep_image(filename)

  # Make a prediction
  pred = model.predict(tf.expand_dims(img, axis=0))

  # Get the predicted class
  pred_class = class_names[int(tf.round(pred))]

  # Plot the image and predicted class
  plt.imshow(img)
  plt.title(f"Prediction: {pred_class}")
  plt.axis(False);
```

```
# Test our model on a custom image
pred_and_plot(model_7, "03-steak.jpeg")
```