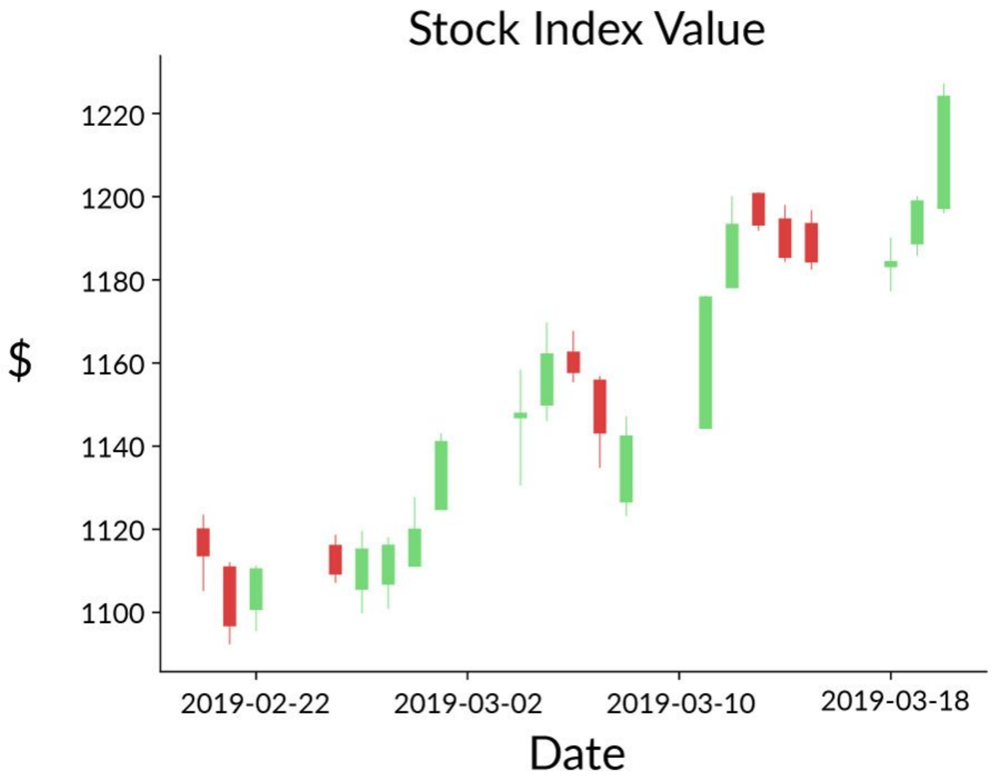
Neural networks and deep learning

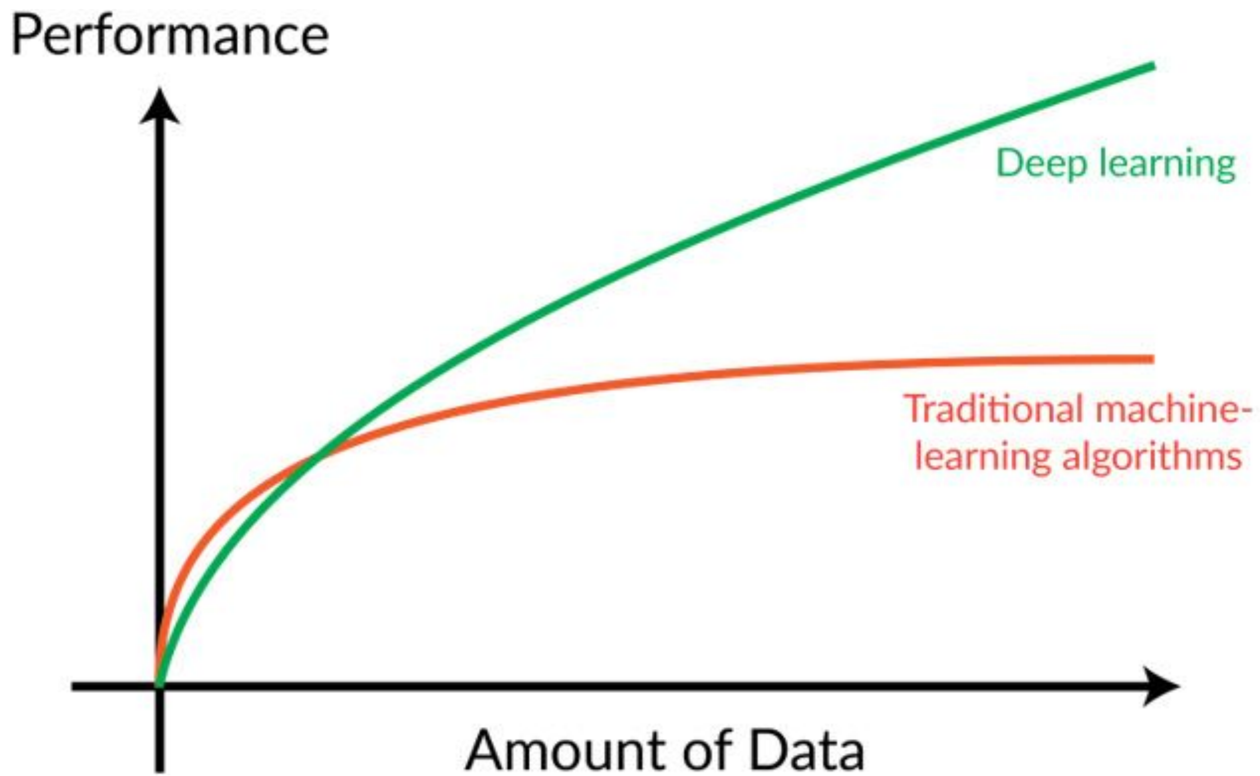
LEARNING VOYAGE

Introduction



Advantages of ANNs over Traditional Machine Learning Algorithms

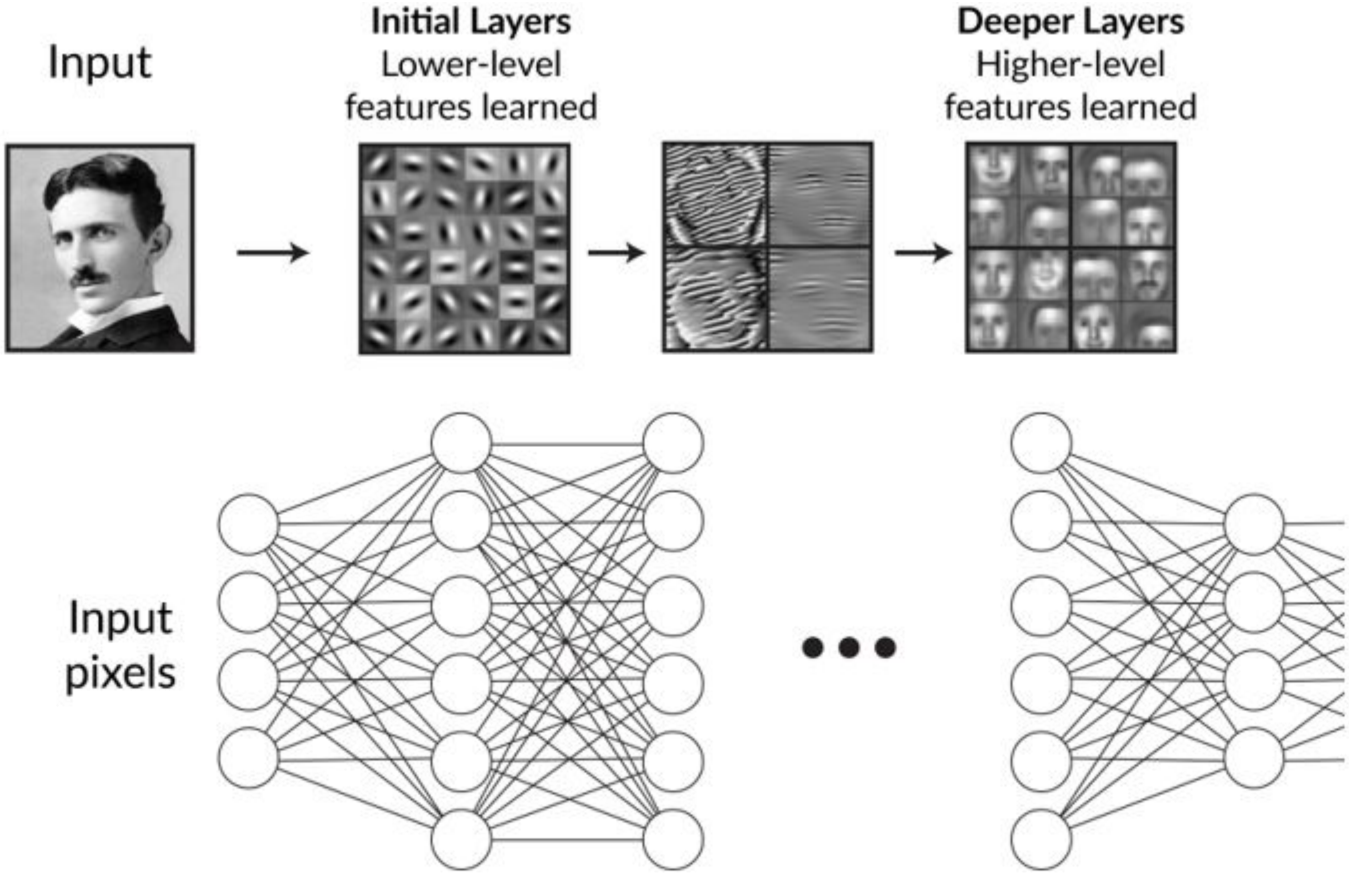
- The best performance
- Scale effectively with data
- No need for feature engineering
- Adaptable and transferable


Performance
Deep learning
Traditional machine-
learning algorithms
Amount of Data

Advantages of Traditional Machine Learning Algorithms over ANNs

Hierarchical Data Representation

- One reason that ANNs are able to perform so well is that a large number of layers allows the network to learn representations of the data at many different levels.
- At lower levels of the model, simple features are learned, such as edges and gradients, as can be seen by looking at the features that were learned in the initial layers.


Input
Initial Layers
Lower-level
features learned
Deeper Layers
Higher-level
features learned
Input
pixels

Neural networks and deep learning

This lesson covers

- Convolutional neural networks for image classification
- TensorFlow and Keras — frameworks for building neural networks
- Using pre-trained neural networks
- Internals of a convolutional neural network
- Training a model with transfer learning
- Data augmentations — the process of generating more training data

Fashion classification

The plan for our project is the following:

- First, we'll download the dataset and use a pre-trained model to classify images.
- Then we'll talk about neural networks, see how they work internally.
- After that, we'll take the pre-trained neural network and adjust it for solving our tasks.
- Finally, we'll expand our dataset by generating many many more images from the images we have.

GPU vs CPU

- Training a neural network is a computationally demanding process, and it requires powerful hardware to make it faster.
- To speed up training, we usually use GPUs – graphical processing units, or, simply, graphic cards.
- For this lesson, a GPU is not required, You can do everything on your laptop, but without a GPU, it will be approximately 8 times slower than with a GPU.

Downloading the clothing dataset

- For this project, we need a dataset of clothes.
- We will use a subset of the clothing dataset[3], which contains around 3,800 images of ten different classes.
- The data is available in a GitHub repository. Let’s clone it:

```
!git clone
https://github.com/fenago/clothing-dataset-small-master.git
```

The dataset is already split into train, validation and test

Each of these folders has 10 subfolders: one subfolder for each type of clothes

Images in the dataset are organized in subfolders


0a7e5fe0-d592-40e6-b9b8-75a...
0ad5bfb5-0f2b-4396-8c05-39c...
0c2eb9ff-7f26-492d-9957-0d8...
0c99f0b4-3a0d-4d24-bfdd-e9e...
0c224954-0e0f-4caa-82c8-cf95...
0ccc318a-7d69-4d7f-a442-aac...
0db5a848-2066-436f-bd21-8b3...
0e3d71f8-7677-4cd4-ba24-478...
0e27351a-13d0-41a6-b731-409...
0e684087-83bf-4153-90f4-6f7...
0fe5eeb6-316f-4f60-b604-8a1...
1a08f33a-2ff4-4fb8-920b-8ff5...
1c9a6bc0-d29e-4e2b-8d00-d19...
1ca03195-b1e8-4c47-85de-81a...
1caba263-088a-448a-be6d-300...
1d42c614-19d5-4515-8ef9-0bf...
1d226430-72e6-4be7-998e-604...
1d407629-87e5-4702-b9fc-e3b...

TensorFlow and Keras

- If you use your laptop with Anaconda, or run the code somewhere else, you need to install TensorFlow — a library for building neural networks.

- Use Pip to do it:

```
!pip install tensorflow
```

TensorFlow and Keras

- We begin by importing NumPy and MatplotLib:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

- Next, import TensorFlow and Keras:

```
import tensorflow as tf
from tensorflow import keras
```

Loading images

- There's a special function in Keras for loading images. It's called load_img.
- Let's import it:

```
from tensorflow.keras.preprocessing.image import load_img
```

LOADING IMAGES

- Let’s use this function to take a look at one of the images:

```
path =
'/home/jovyan/clothing-dataset-small/train/t-shirt'
name = '5f0a3fa0-6a3d-4b68-b213-72766a643de7.jpg'
fullname = path + '/' + name
load_img(fullname)
```

```python
path = './clothing-dataset-small/train/t-shirt'
name = '5f0a3fa0-6a3d-4b68-b213-72766a643de7.jpg'
fullname = path + '/' + name
load_img(fullname)
```

- After executing the cell, we should see an image of a T-shirt

- To resize the image, specify the target_size parameter:

```
load_img(fullname, target_size=(299, 299))
```

- As a result, the image becomes square and a bit squashed

Convolutional neural networks

- Neural networks is a class of machine learning models for solving classification and regression problems.
- Our problem is a classification problem – we need to determine the category of an image.
- However, our problem is special: we’re dealing with images.

Using a pre-trained model

- First, we'll need to import the model itself and some helpful functions:

```
from tensorflow.keras.applications.xception
import Xception
from tensorflow.keras.applications.xception
import preprocess_input
from tensorflow.keras.applications.xception
import decode_predictions
```

Using a pre-trained model

- Let’s load this model:

```
model = Xception(
    weights='imagenet',
    input_shape=(299, 299, 3)
)
```

Using a pre-trained model

- First, we load it using load_img function:

```
img = load_img(fullname, target_size=(299, 299))
```

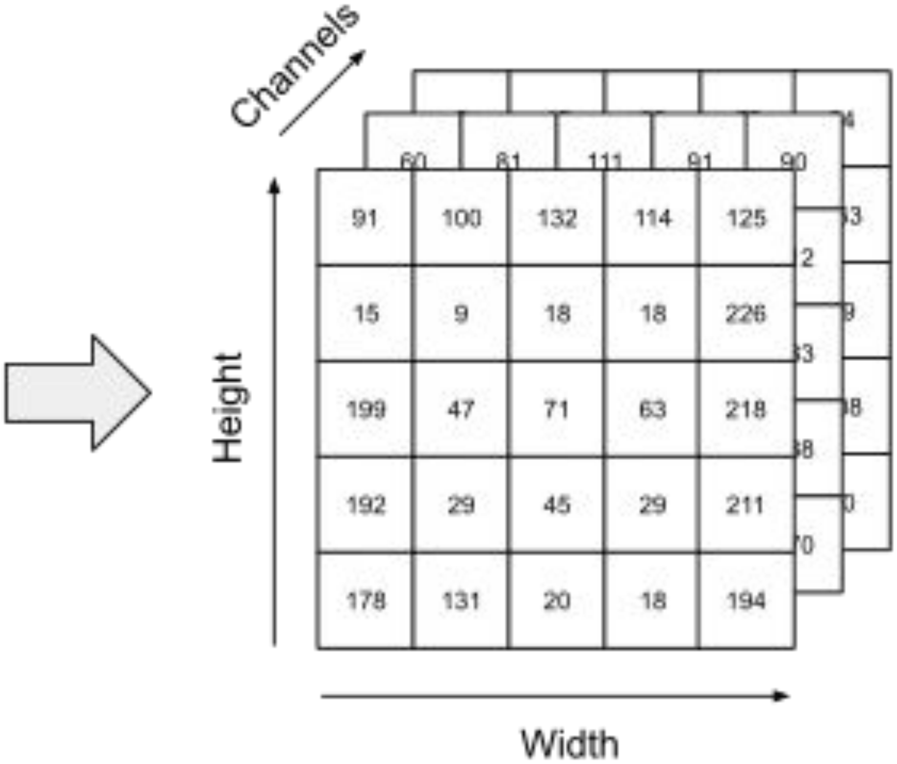
- The img variable is an Image object, which we need to convert to a NumPy array, It's easy to do:

```
x = np.array(img)
```

- This array should have the same shape as the image, Let's check it:

```
x.shape
```

Using a pre-trained model



Using a pre-trained model

- Since we just one image, we need to create a batch with this single image:

```
X = np.array([x])
```

- : If we had several images, for example, x, y and z, we'd write:

```
X = np.array([x, y, z])
```

- Let's check its shape:

```
X.shape
```

Using a pre-trained model

- Before we can apply the model to our image, we need to prepare it.
- We do it with the preprocess_input function:

```
X = preprocess_input(X)
```

Getting predictions

- To apply the model, use the predict method:

```
pred = model.predict(X)
```

- Let’s take a look at this array:

```
pred.shape
```

Getting predictions

- Luckily, there's a function decode_predictions that we can use to decode the prediction into meaningful class names:

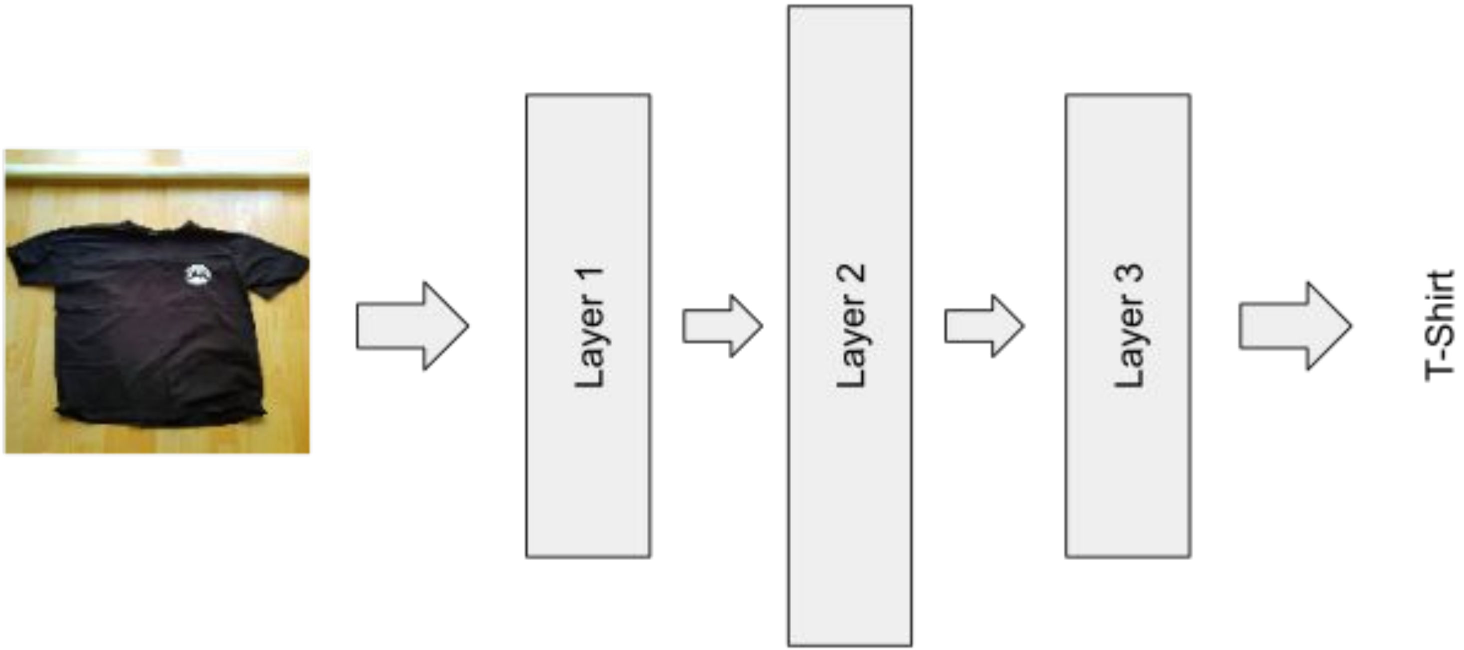
```
decode_predictions(pred)
```

Getting predictions

- It shows the top-5 most likely classes for this image:

```
[[('n02667093', 'abaya', 0.028757658),
  ('n04418357', 'theater_curtain', 0.020734021),
  ('n01930112', 'nematode', 0.015735716),
  ('n03691459', 'loudspeaker', 0.013871926),
  ('n03196217', 'digital_clock', 0.012909736)]]
```

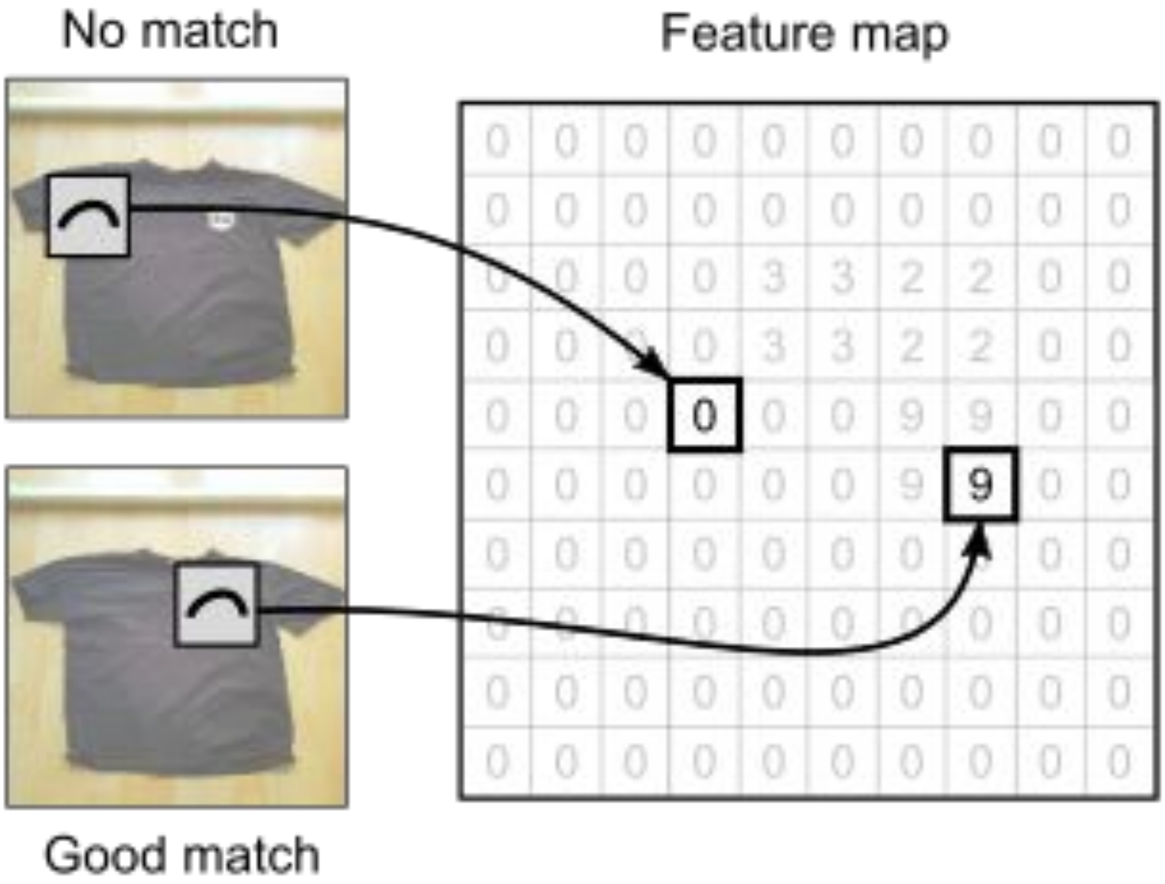
Internals of the model



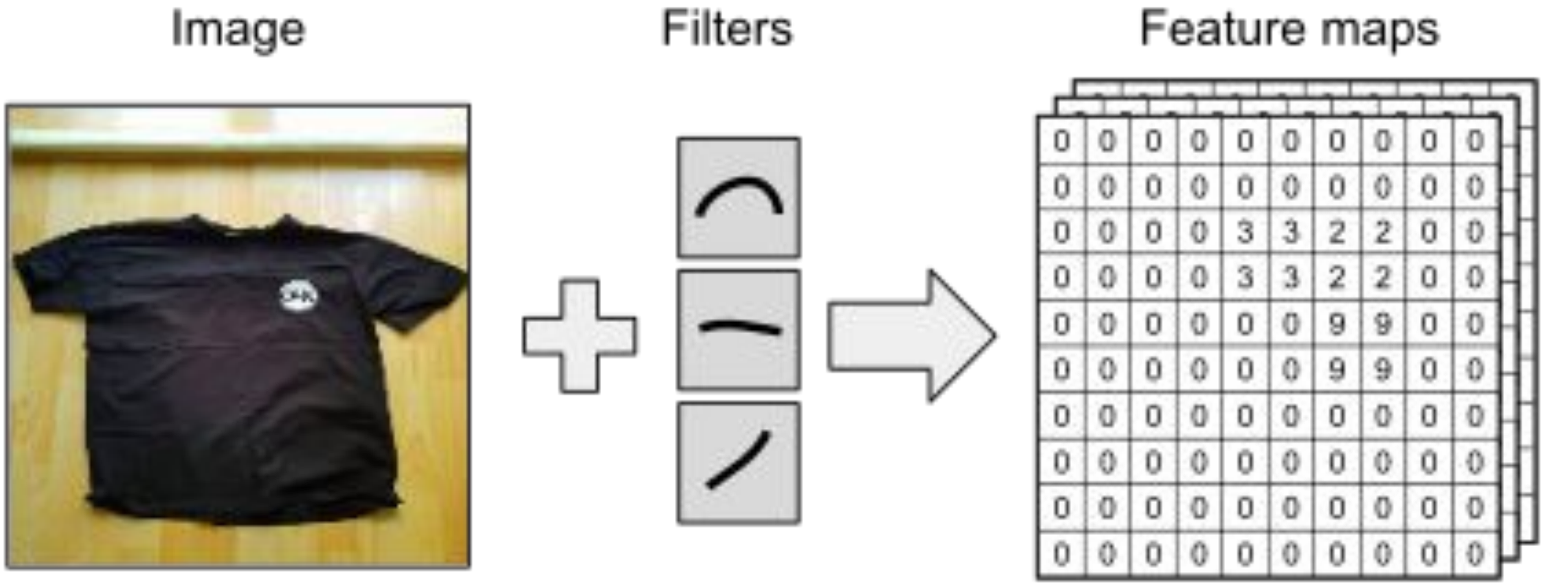
Convolutional layers

- Even though "convolutional layer" sounds complicated, it's nothing more than a set of filters – small "images" with simple shapes like stripes

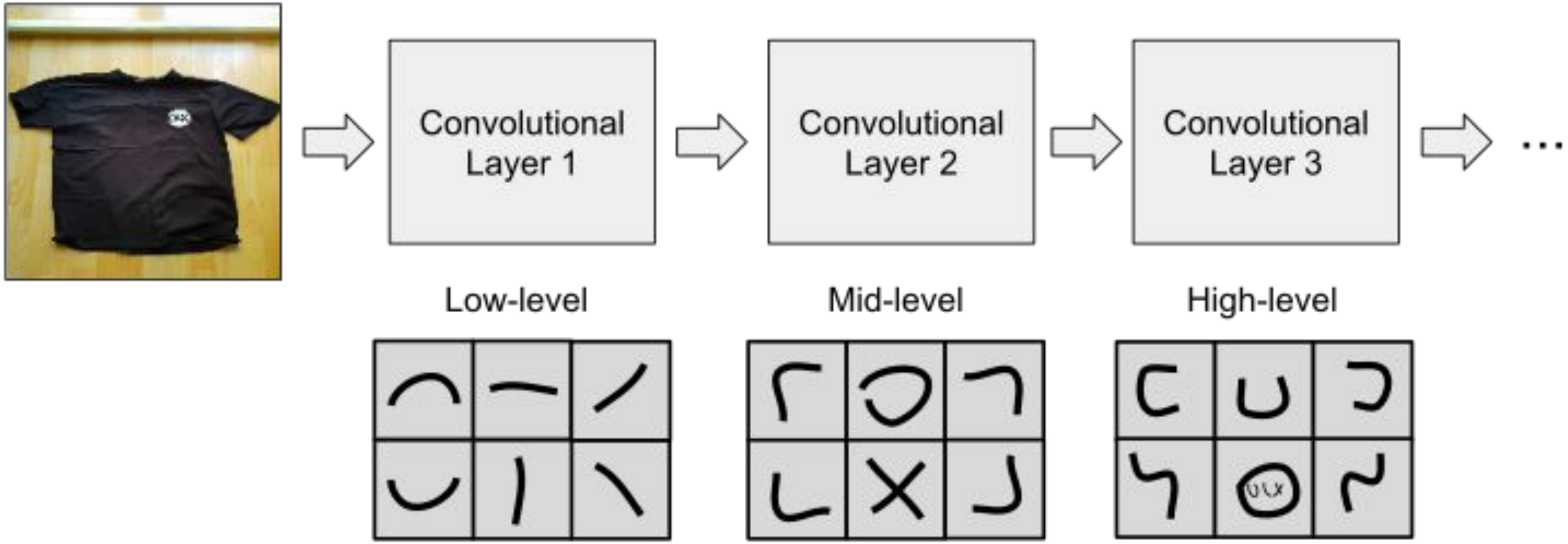
Convolutional layers


No match
Feature map
0
9
Good match

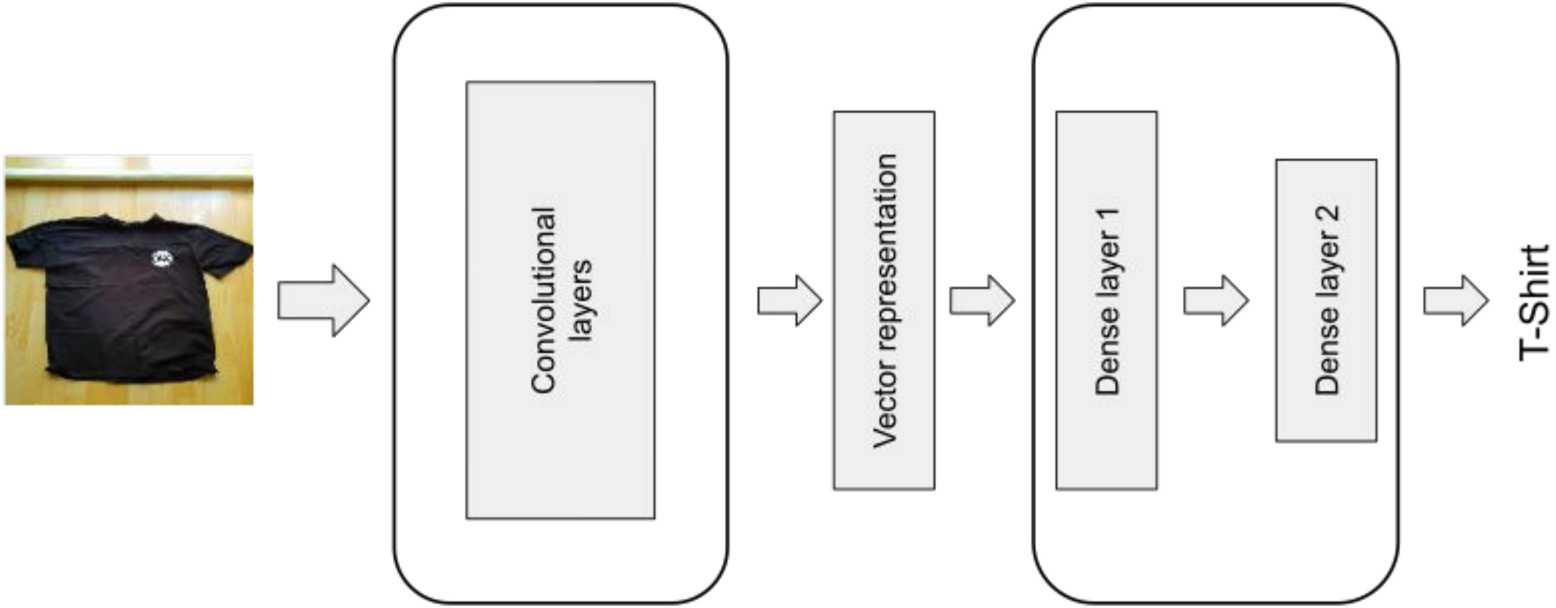
Convolutional layers



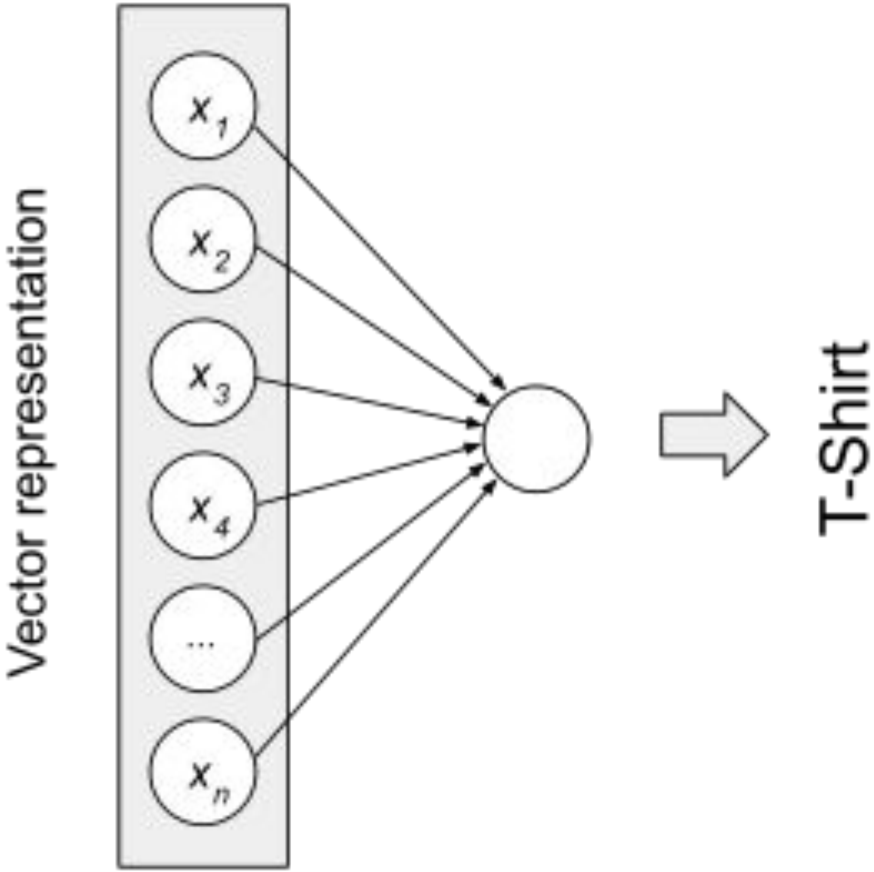
Convolutional layers



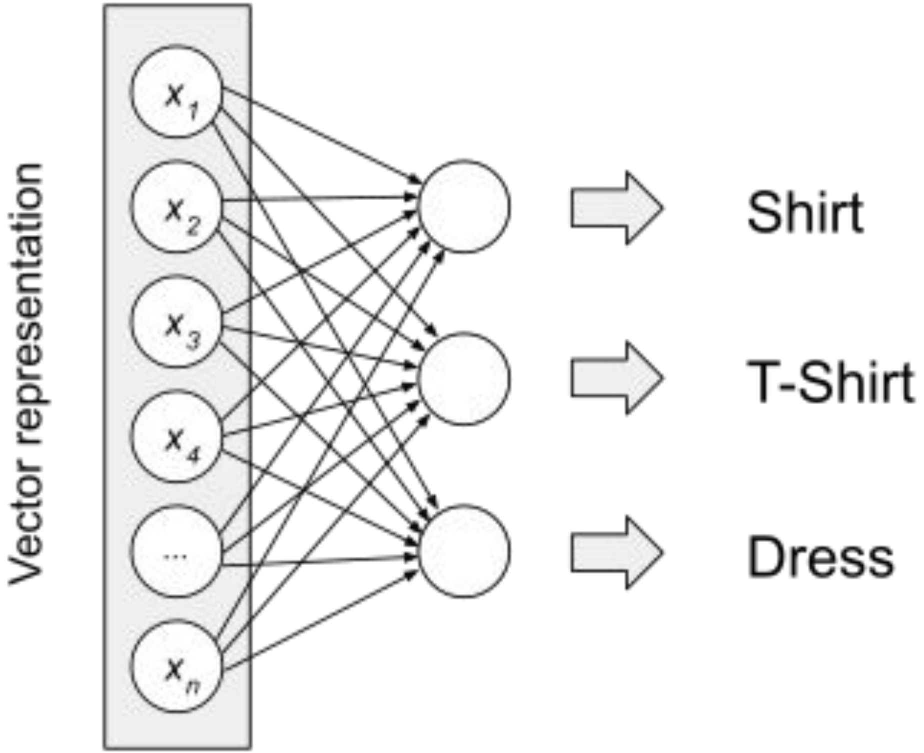
Dense layers

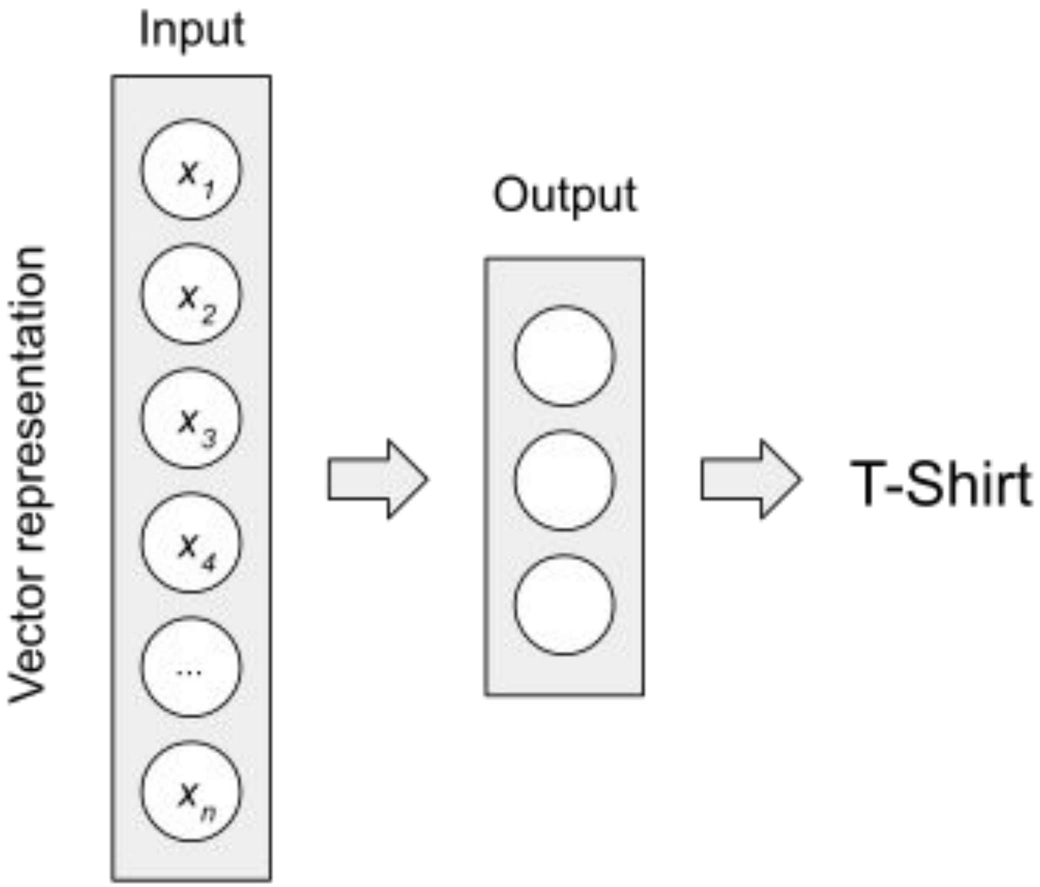


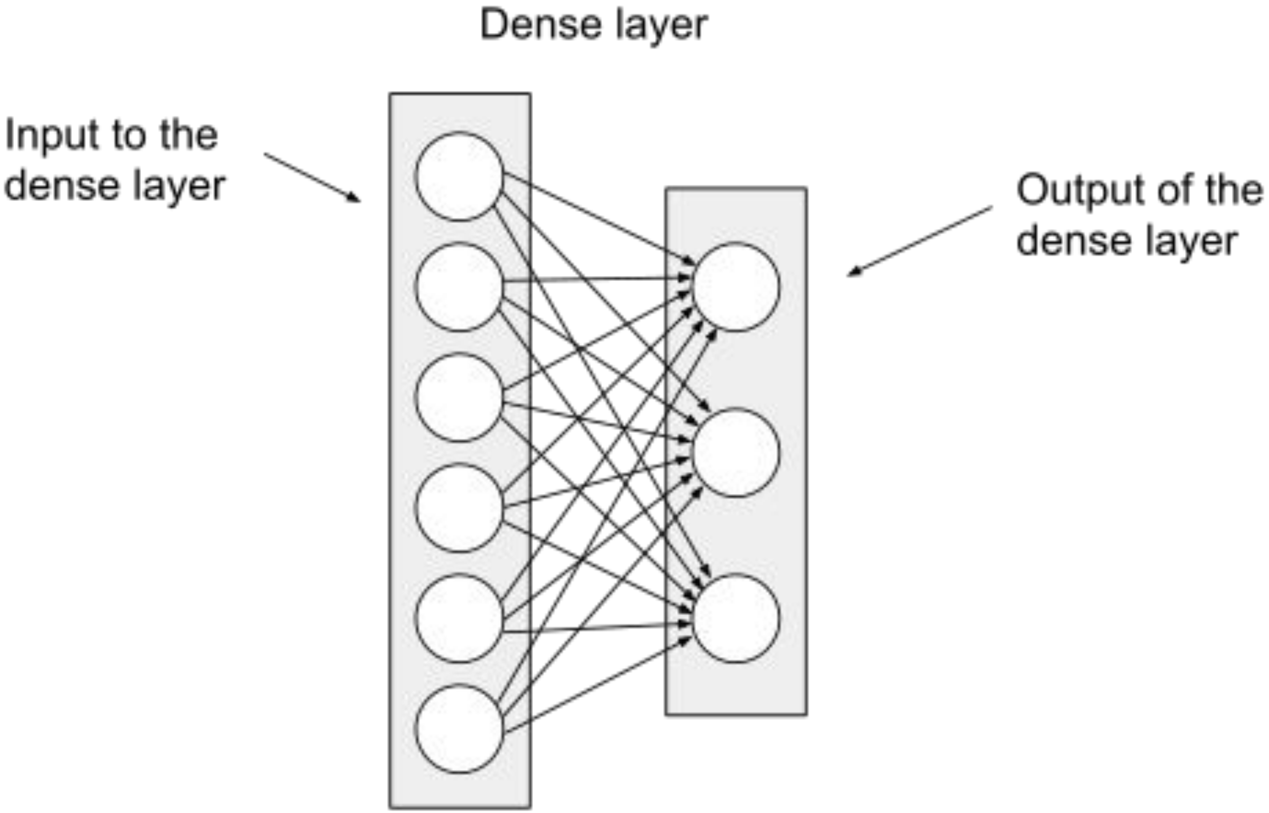
Dense layers

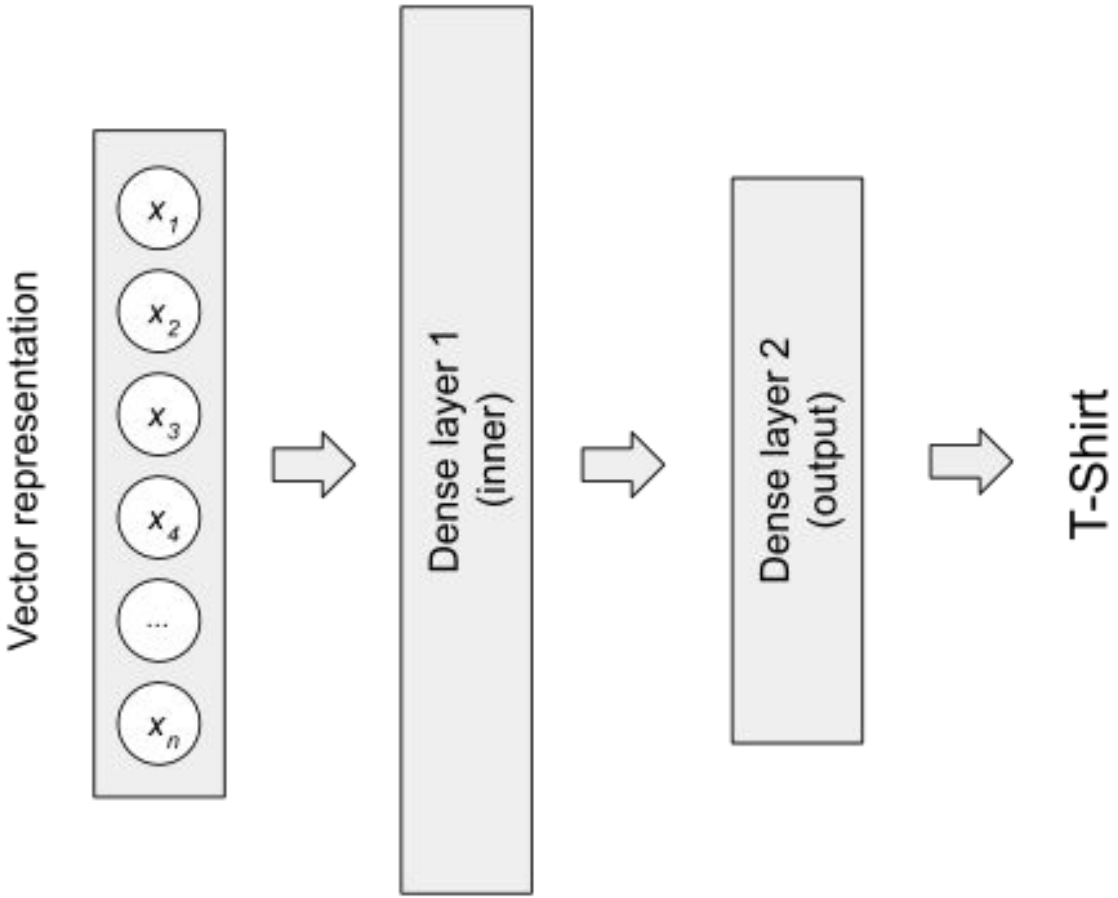


Dense layers

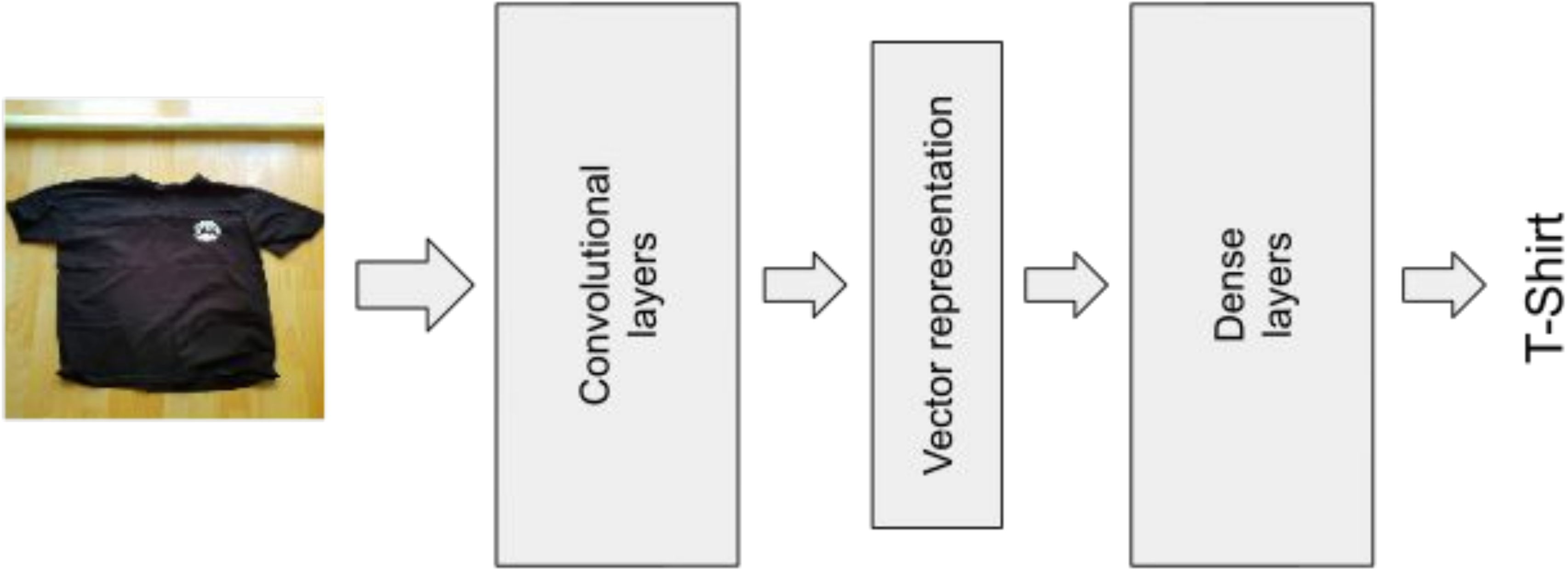



Input
Vector representation
x_1
x_2
x_3
x_4
...
x_n
Output
T-Shirt


Dense layer
Input to the dense layer
Output of the dense layer


Vector representation
x_1
x_2
x_3
x_4
...
x_n
Dense layer 1
(inner)
Dense layer 2
(output)
T-Shirt

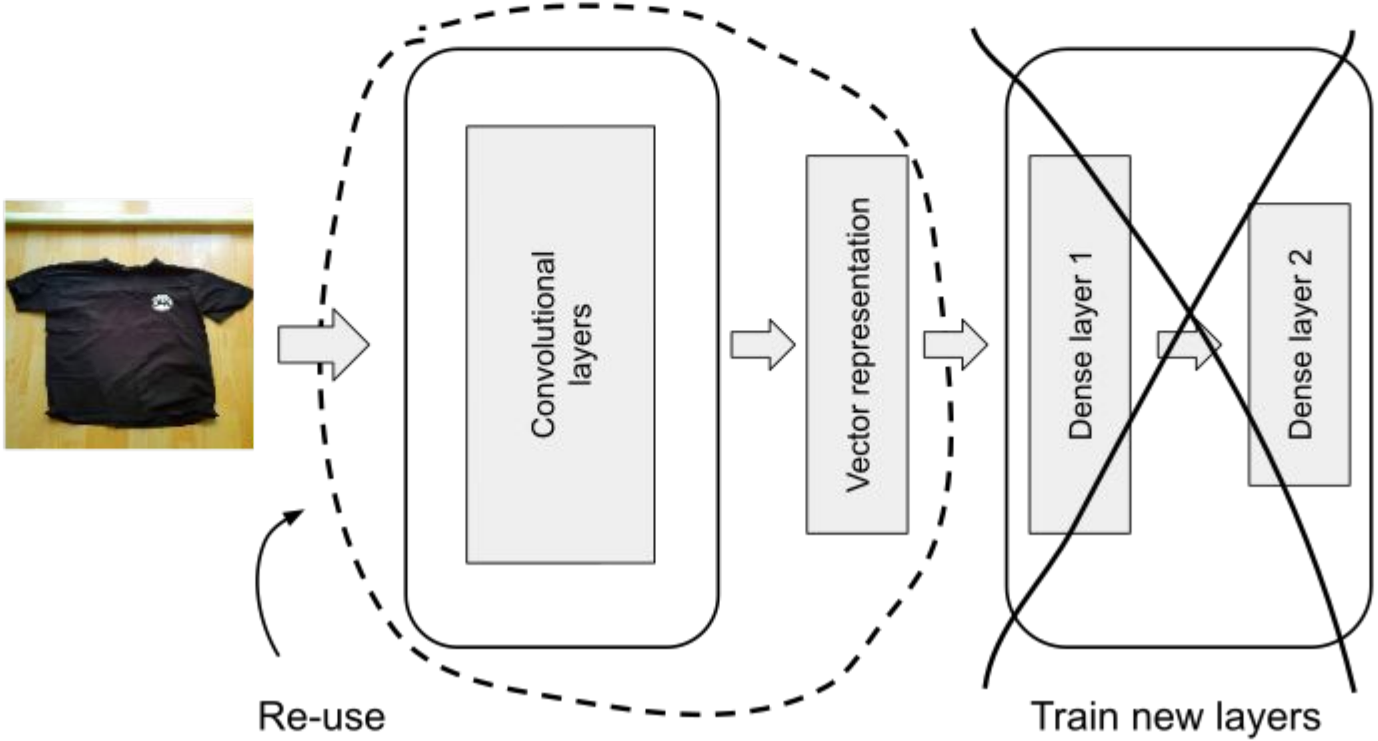
Dense layers



Training the model

- Training a convolutional neural network takes a lot of time and requires a lot of data.
- But there's a shortcut: we can use transfer learning, an approach where we take a pre-trained model and adapt it to our problem.

Transfer learning



Loading the Data

- Keras comes with a solution – ImageDataGenerator.
- Instead of loading the entire dataset into memory, it loads the images from disk in small batches, Let's use it:

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

train_gen = ImageDataGenerator(
    preprocessing_function=preprocess_input
)
```

Loading the data

- We have a generator now, so we just need to point it to the directory with the data.

- For that, use the flow_from_directory method:

```
train_ds = train_gen.flow_from_directory(
    "clothing-dataset-small/train",
    target_size=(150, 150),
    batch_size=32,
)
```

LOADING THE DATA

- When we execute the cell, it informs us how many images are there in the train dataset and how many classes:

```
Found 3068 images belonging to 10 classes.
```

Loading the data

- Now we repeat the same process for the validation dataset:

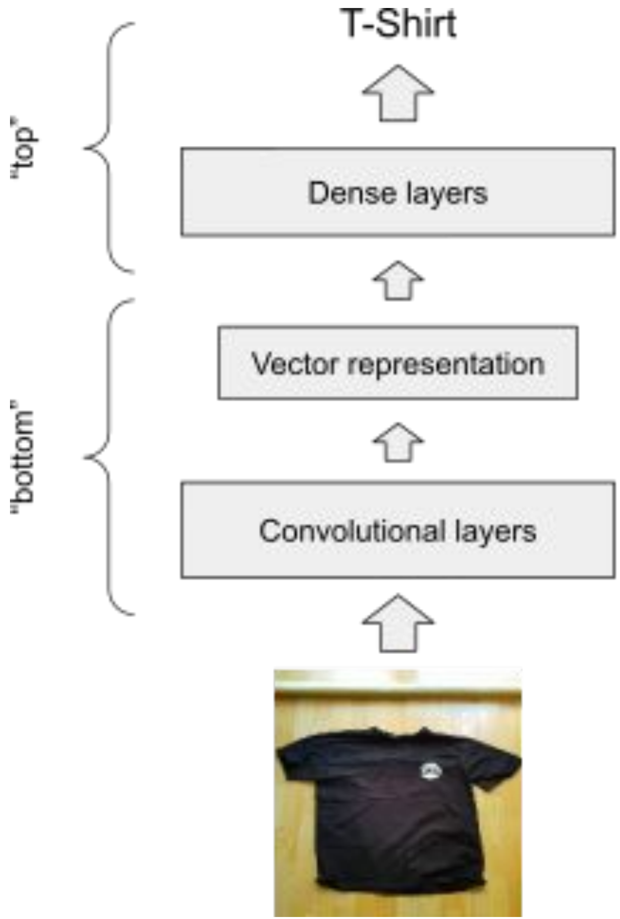
```
validation_gen = ImageDataGenerator(
    preprocessing_function=preprocess_input
)

val_ds = validation_gen.flow_from_directory(
    "clothing-dataset-small/validation",
    target_size=image_size,
    batch_size=batch_size,
)
```

Creating the model

- let’s create the base model:

```
base_model = Xception(
    weights='imagenet',
    include_top=False
    input_shape=(150, 150, 3),
)
```

Creating the model



Creating the model

- We don't want to train the base model: attempting to do it will destroy all the filters.
- So, we "freeze" the the base model by setting the trainable parameter to False:

```
base_model.trainable = False
```

Creating the model

- Now let's build the clothes classification model:

```
inputs = keras.Input(shape=(150, 150, 3))

base = base_model(inputs, training=False)
vector =
keras.layers.GlobalAveragePooling2D()(base)

outputs = keras.layers.Dense(10)(vector)

model = keras.Model(inputs, outputs)
```

Creating the model

- First, we specify the input and the size of the arrays we expect:

```
inputs = keras.Input(shape=(150, 150, 3))
```

- Next, we create the base model:

```
base = base_model(inputs, training=False)
```

Creating the model

- The result is base, which is a functional component (like base_model) that we can combine with other components.

- We use it as the input to the next layer:

```
vector =
keras.layers.GlobalAveragePooling2D()(base)
```

Creating the model

- It may be a bit confusing because we create a layer and immediately connect it to base.
- We can rewrite it to make it simpler to understand:

```
pooling = keras.layers.GlobalAveragePooling2D()
vector = pooling(base)
```

Creating the model

- Another functional component that we connect to the next layer – a dense layer:

```
outputs = keras.layers.Dense(10)(vector)
```

Creating the model

- In our case, the data comes in inputs and goes out of outputs.
- We just need to do one final step: wrap both inputs and outputs into a Model class:

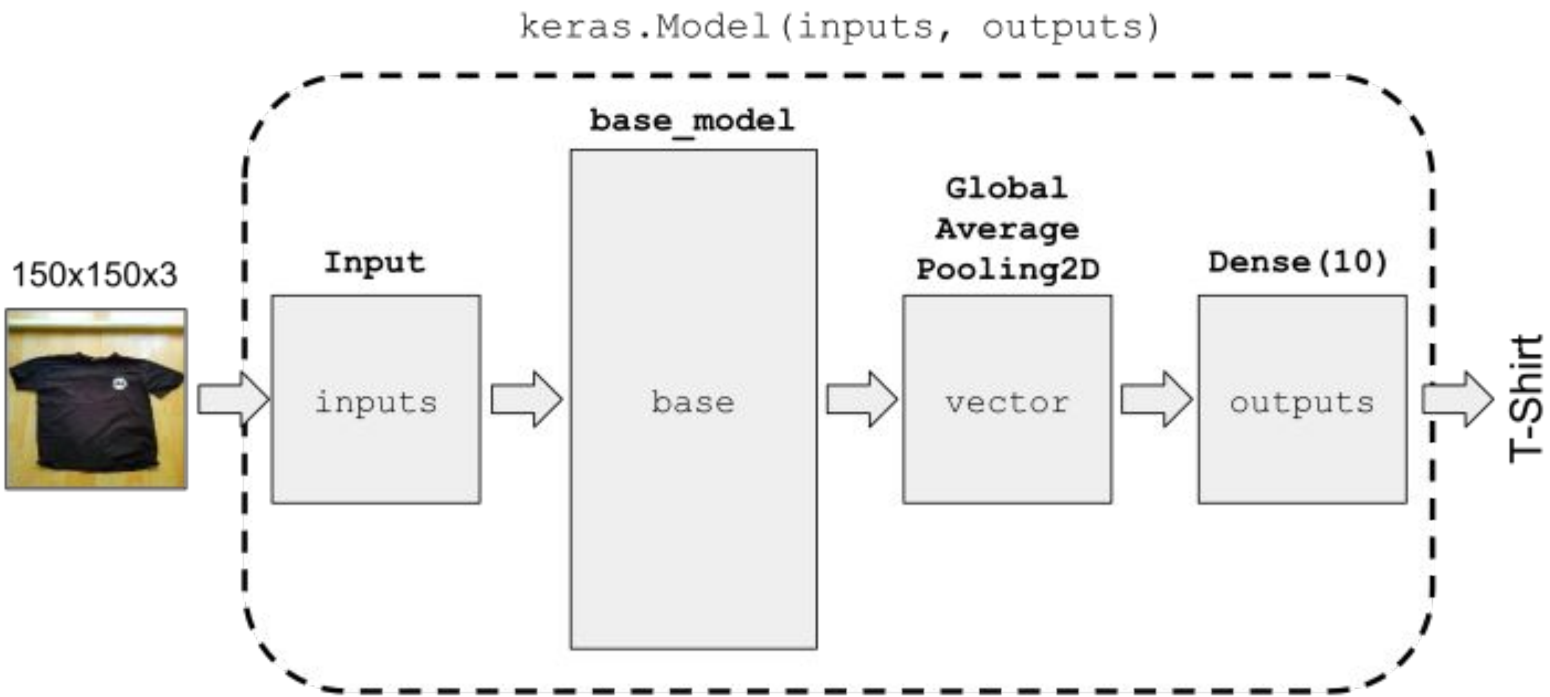
```
model = keras.Model(inputs, outputs)
```

Creating the model

```
inputs = keras.Input(shape=(150, 150, 3))
base = base_model(inputs, training=False)
vector = keras.layers.GlobalAveragePooling2D()(base)
outputs = keras.layers.Dense(10)(vector)
model = keras.Model(inputs, outputs)
```

T-Shirt

Creating the model



Training the model

- Let's create it:

```
learning_rate = 0.01
optimizer = keras.optimizers.Adam(learning_rate)
```

Training the model

- We need to classify clothes into 10 different classes, so we'll use the categorical cross-entropy loss:

```
loss =
keras.losses.CategoricalCrossentropy(from_logits=
True)
```

Training the model

- In this case, we explicitly tell the network to output probabilities: softmax is similar to sigmoid, but for multiple classes.
- Then the output is not “logits” anymore, so we can drop this parameter:

```
loss = keras.losses.CategoricalCrossentropy()
```

Training the model

- Now let's put the optimizer and the loss together.
- For that, we'll use the compile method of our model:

```
model.compile(
    optimizer=optimizer,
    loss=loss,
    metrics=["accuracy"]
)
```

Training the model

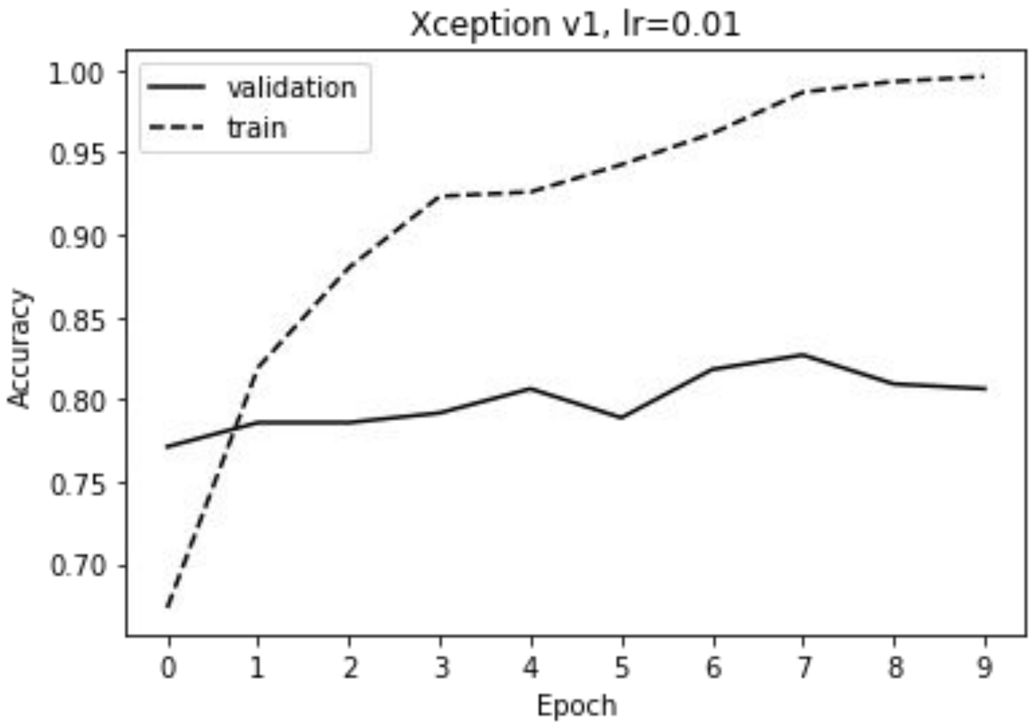
- Our model is ready for training! To do it, use the fit method:

```
model.fit(train_ds, epochs=10,
validation_data=val_ds)
```

- When we start training, Keras informs us about the progress:

```
Train for 96 steps, validate for 11 steps
Epoch 1/10
96/96 [==============================] - 22s 227ms/step -
loss: 1.2372 - accuracy: 0.6734 - val_loss: 0.8453 -
val_accuracy: 0.7713
Epoch 2/10
96/96 [==============================] - 16s 163ms/step -
loss: 0.6023 - accuracy: 0.8194 - val_loss: 0.7928 -
val_accuracy: 0.7859
...
Epoch 10/10
96/96 [==============================] - 16s 165ms/step -
loss: 0.0274 - accuracy: 0.9961 - val_loss: 0.9342 -
val_accuracy: 0.8065
```

Training the model



"Complete Exercise "

Adjusting the learning rate

```
def make_model(learning_rate):
    base_model = Xception(
        weights='imagenet',
        input_shape=(150, 150, 3),
        include_top=False
    )

    base_model.trainable = False

    inputs = keras.Input(shape=(150, 150, 3))

    base = base_model(inputs, training=False)
    vector = keras.layers.GlobalAveragePooling2D()(base)

    outputs = keras.layers.Dense(10)(vector)

    model = keras.Model(inputs, outputs)

    optimizer = keras.optimizers.Adam(learning_rate)
    loss = keras.losses.CategoricalCrossentropy(from_logits=True)

    model.compile(
        optimizer=optimizer,
        loss=loss,
        metrics=["accuracy"],
    )

    return model
```

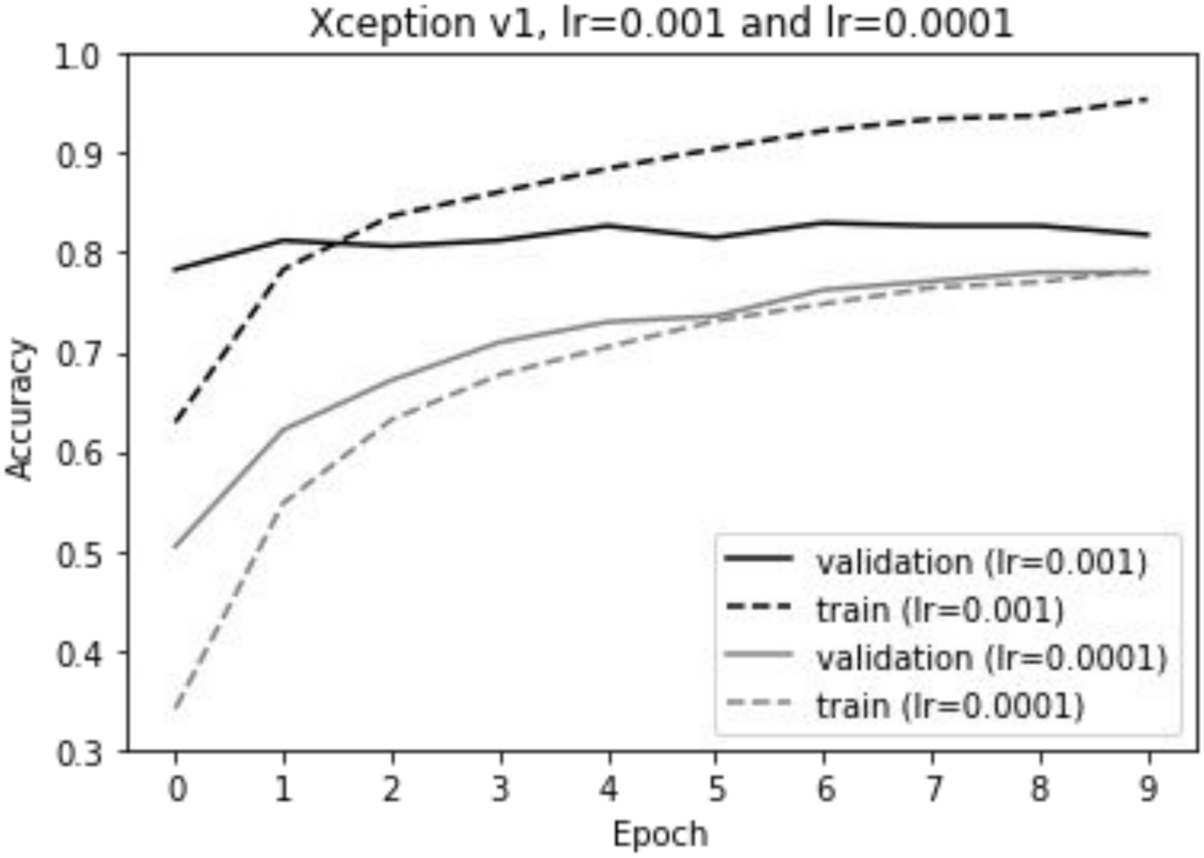
Adjusting the learning rate

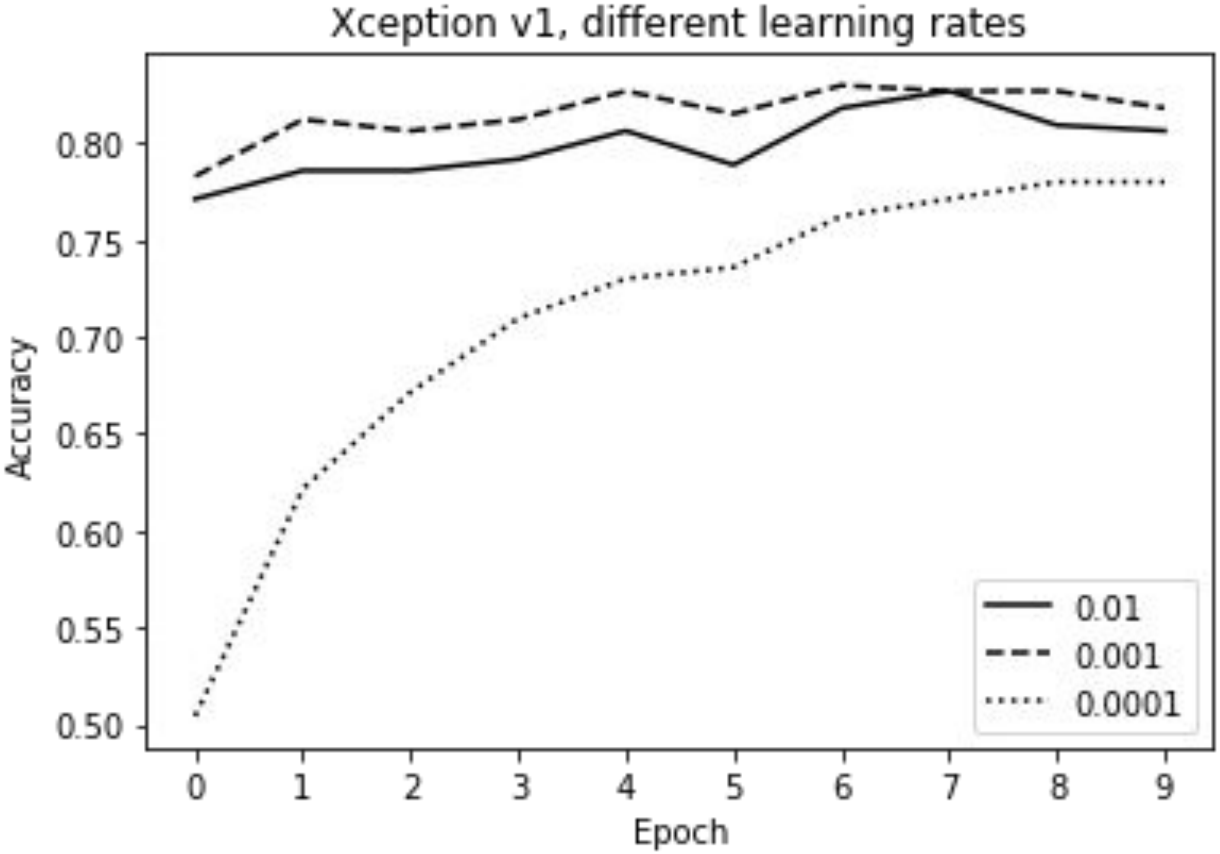
- We've tried 0.01, so let's try 0.001:

```
model = make_model(learning_rate=0.001)
model.fit(train_ds, epochs=10,
validation_data=val_ds)
```

- We can also try even smaller value of 0.0001:

```
model = make_model(learning_rate=0.0001)
model.fit(train_ds, epochs=10,
validation_data=val_ds)
```


Xception v1, lr=0.001 and lr=0.0001
Accuracy
Epoch
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
1
2
3
4
5
6
7
8
9
validation (lr=0.001)
train (lr=0.001)
validation (lr=0.0001)
train (lr=0.0001)


Xception v1, different learning rates
Accuracy
0.50
0.55
0.60
0.65
0.70
0.75
0.80
Epoch
0
1
2
3
4
5
6
7
8
9
0.01
0.001
0.0001

Adjusting the learning rate

- For the learning rate of 0.001, the best accuracy is 83%

Learning rate	0.01	0.001	0.0001
Validation accuracy	82.7%	83.0%	78.0%

Saving the model and checkpointing

- Once the model is trained, we can save it using the save_weights method:

```
model.save_weights('xception_v1_model.h5',
save_format='h5')
```

Saving the model and checkpointing

- Keras has a special class for doing it: ModelCheckpoint, Let's use it:

```
checkpoint = keras.callbacks.ModelCheckpoint(
  "xception_v1_{epoch:02d}_{val_accuracy:.3f}.h5",
    save_best_only=True,
    monitor="val_accuracy"
)
```

- The first parameter is a template for the filename, Let's take a look at it again:

```
"xception_v1_{epoch:02d}_{val_accuracy:.3f}.h5"
```

Saving the model and checkpointing

- We can use it by passing it to the callbacks argument of the fit method:

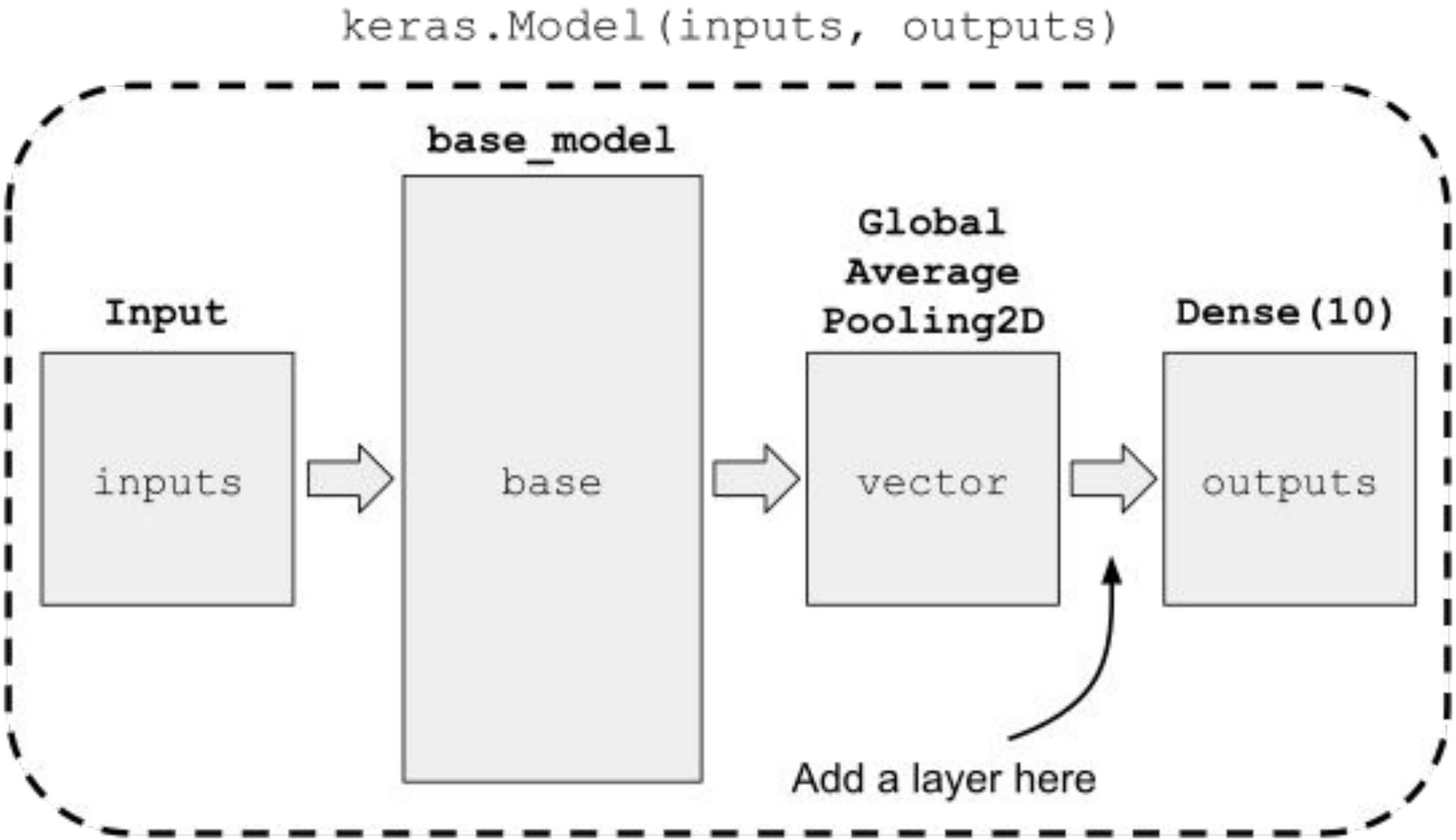
```
model = make_model(learning_rate=0.001)

model.fit(
    train_ds,
    epochs=10,
    validation_data=val_ds,
    callbacks=[checkpoint]
)
```

Saving the model and checkpointing

Adding more layers

- we trained a model with one dense layer:

```
inputs = keras.Input(shape=(150, 150, 3))

base = base_model(inputs, training=False)
vector =
keras.layers.GlobalAveragePooling2D()(base)

outputs = keras.layers.Dense(10)(vector)

model = keras.Model(inputs, outputs)
```

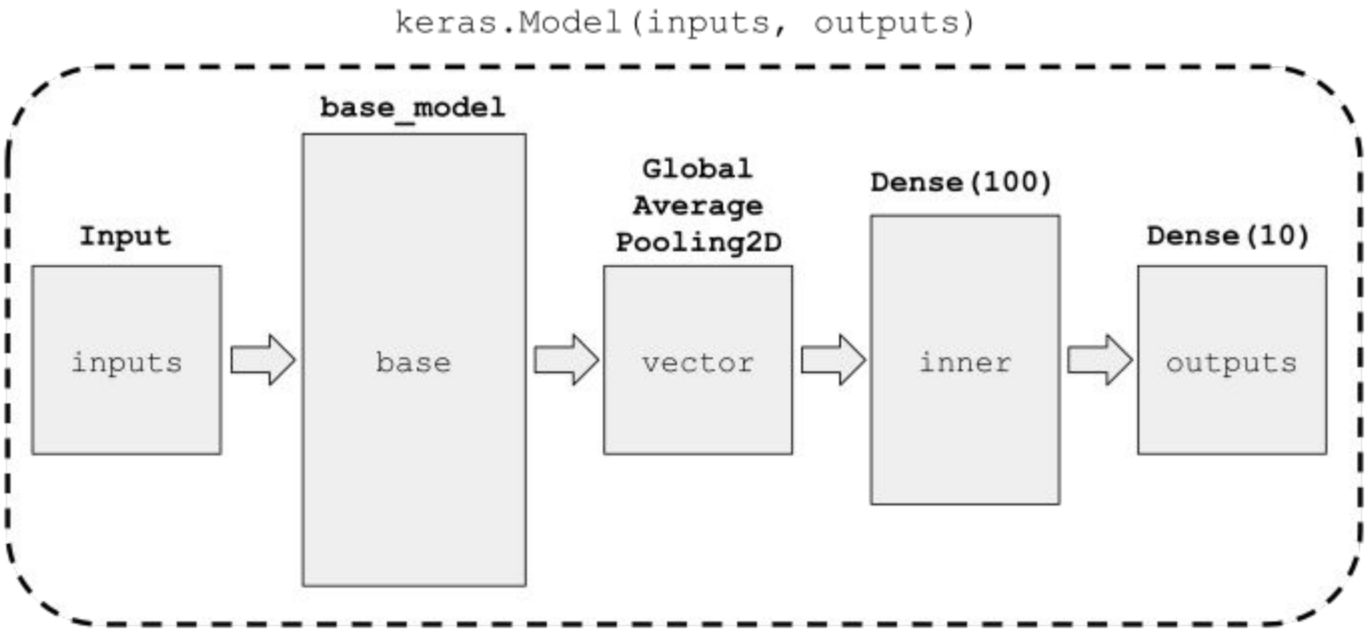

keras.Model(inputs, outputs)
base_model
Input
inputs
base
Global Average Pooling2D
vector
Dense(10)
outputs
Add a layer here

Adding more layers

- For example, we can add a dense layer of size 100:

```
inputs = keras.Input(shape=(150, 150, 3))
base = base_model(inputs, training=False)
vector = keras.layers.GlobalAveragePooling2D()(base)

inner = keras.layers.Dense(100,
activation='relu')(vector)

outputs = keras.layers.Dense(10)(inner)

model = keras.Model(inputs, outputs)
```

Adding more layers

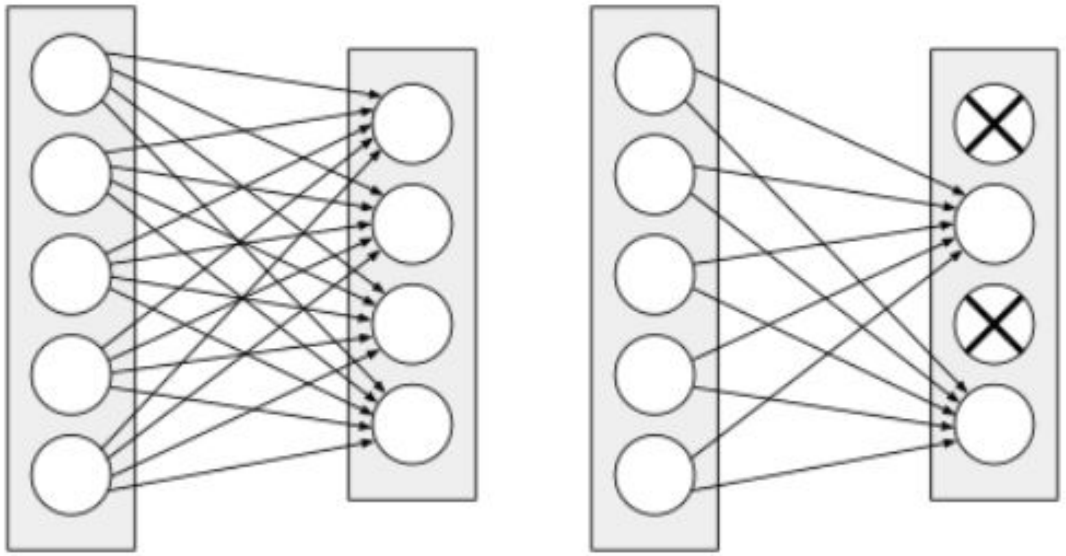


Adding more layers

- Let’s take another look at the line with the new dense layer:

```
inner = keras.layers.Dense(100,
activation='relu')(vector)
```

- Here, we set the activation parameter to “relu”.



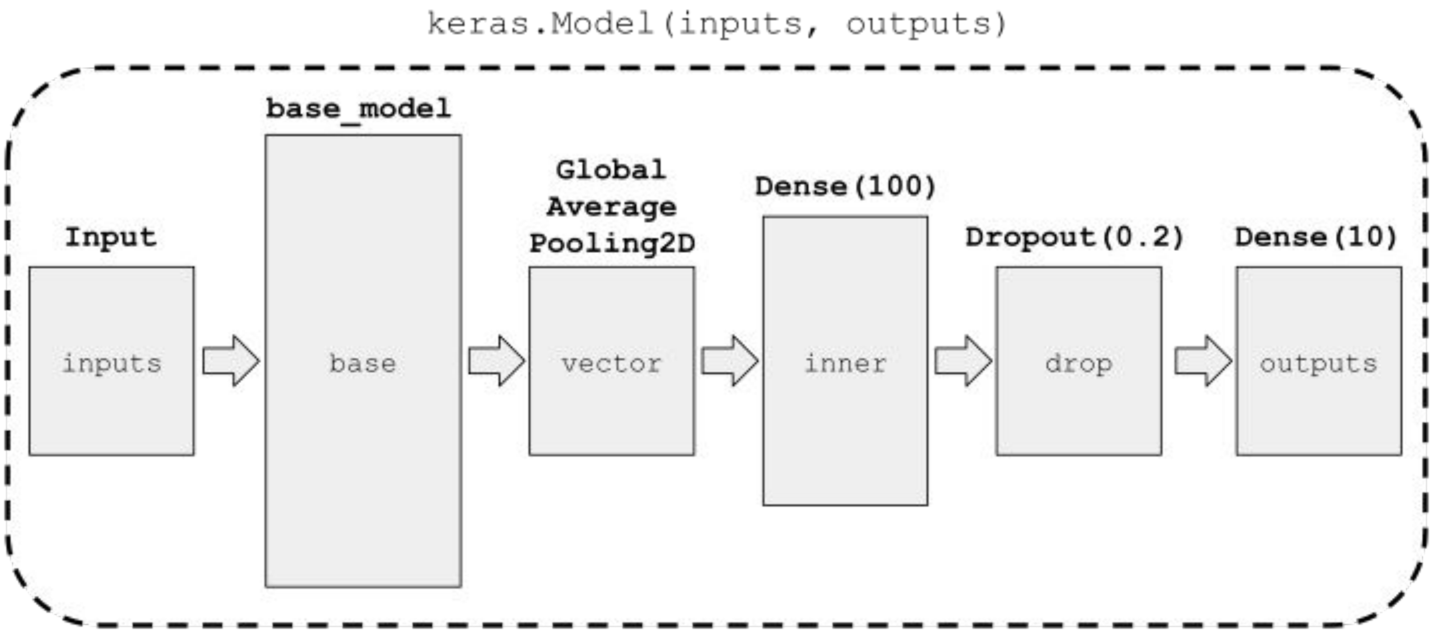
(A) Two dense layers without dropout

(B) Two dense layers with dropout

Regularization and dropout

```
inputs = keras.Input(shape=(150, 150, 3))
base = base_model(inputs, training=False)
vector =
keras.layers.GlobalAveragePooling2D()(base)

inner = keras.layers.Dense(100,
activation='relu')(vector)
drop = keras.layers.Dropout(0.2)(inner)
outputs = keras.layers.Dense(10)(drop)

model = keras.Model(inputs, outputs)
```

Regularization and dropout

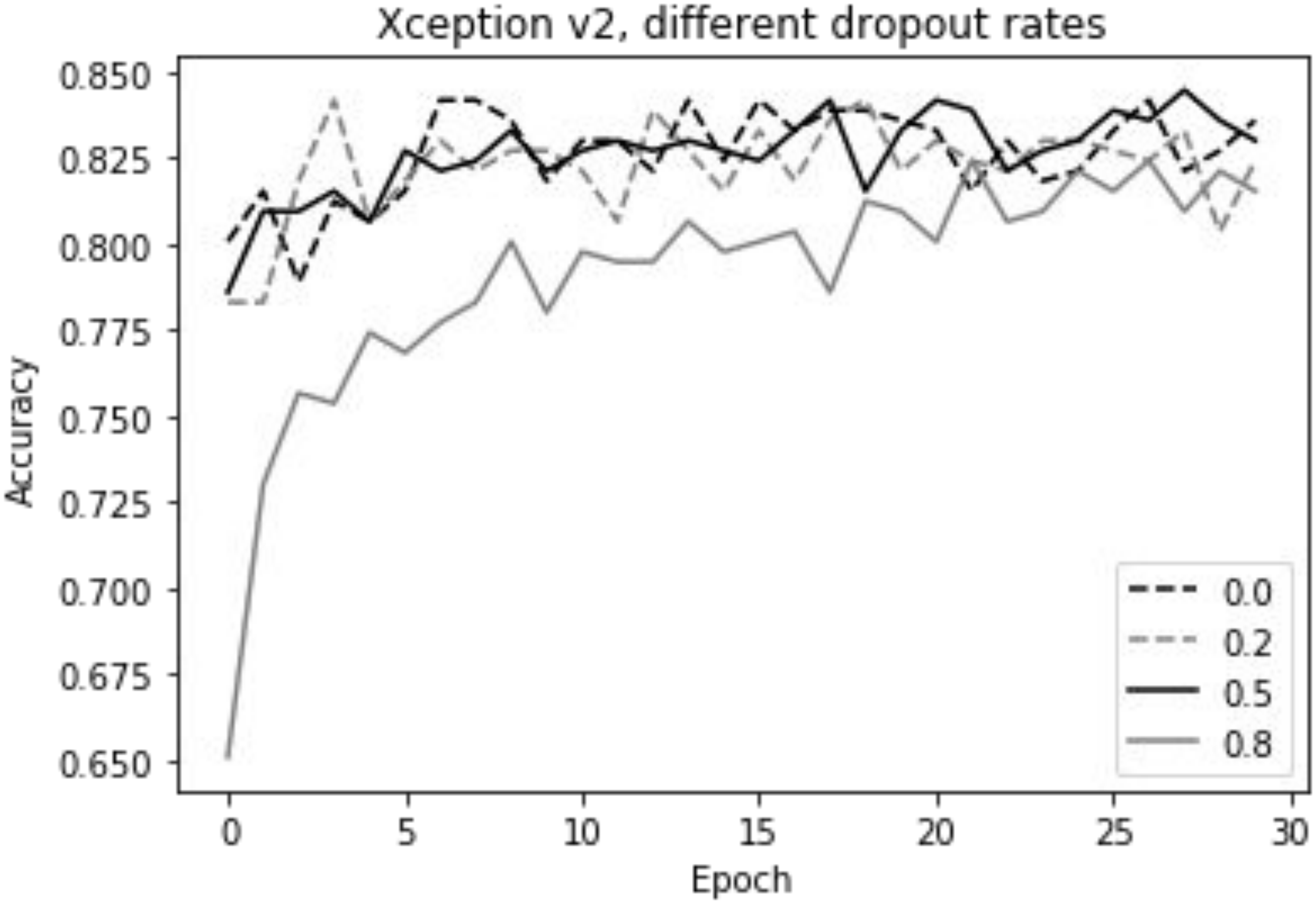


```python
def make_model(learning_rate, droprate):
    base_model = Xception(
        weights='imagenet',
        input_shape=(150, 150, 3),
        include_top=False
    )

    base_model.trainable = False

    inputs = keras.Input(shape=(150, 150, 3))
    base = base_model(inputs, training=False)
    vector = keras.layers.GlobalAveragePooling2D()(base)

    inner = keras.layers.Dense(100, activation='relu')(vector)
    drop = keras.layers.Dropout(droprate)(inner)

    outputs = keras.layers.Dense(10)(drop)

    model = keras.Model(inputs, outputs)

    optimizer = keras.optimizers.Adam(learning_rate)
    loss = keras.losses.CategoricalCrossentropy(from_logits=True)

    model.compile(
        optimizer=optimizer,
        loss=loss,
        metrics=["accuracy"],
    )

    return model
```

Regularization and dropout

- So, let's train it:

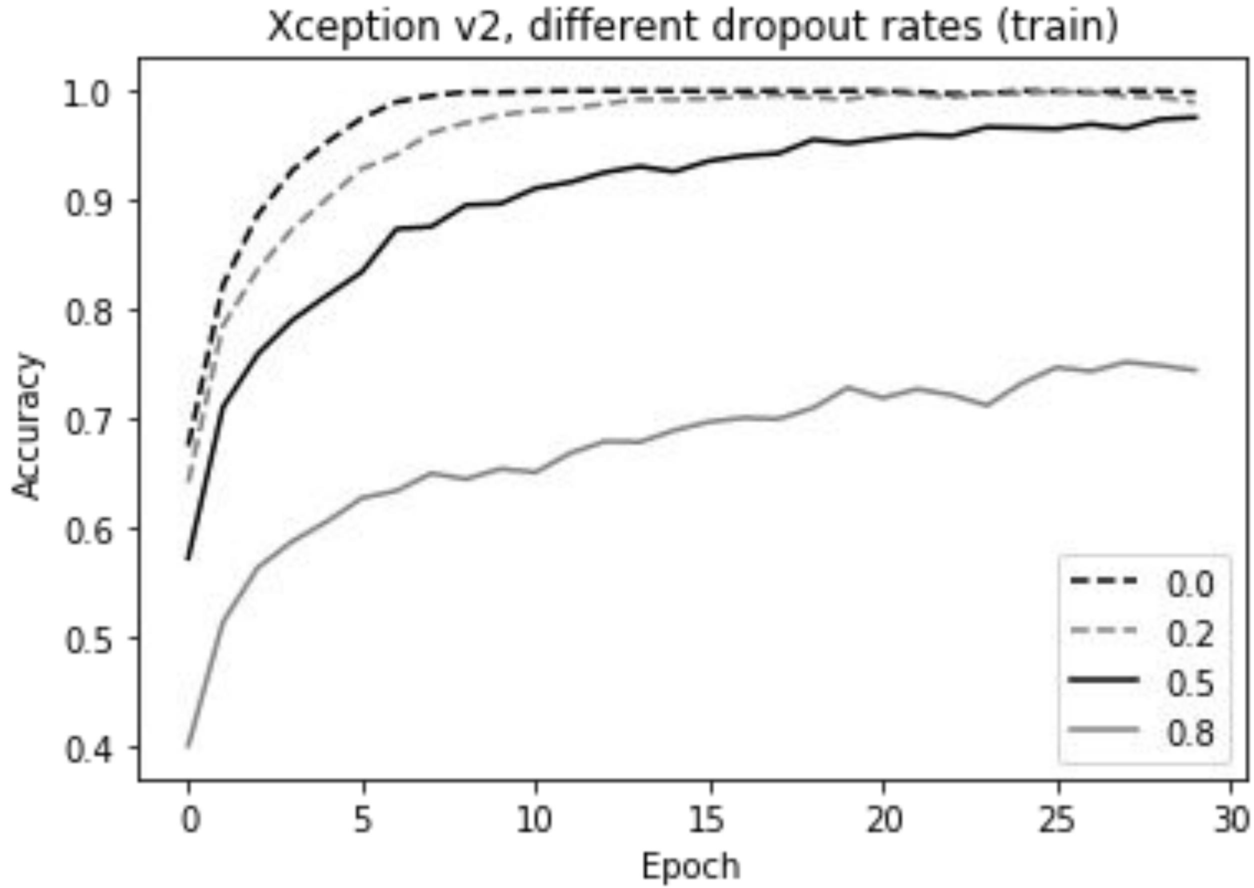
```
model = make_model(learning_rate=0.001,
droprate=0.0)
model.fit(train_ds, epochs=30,
validation_data=val_ds)
```


Xception v2, different dropout rates
0.850
0.825
0.800
0.775
0.750
0.725
0.700
0.675
0.650
Accuracy
0
5
10
15
20
25
30
Epoch
0.0
0.2
0.5
0.8

Regularization and dropout

- The best accuracy we could achieve is 84.5% for the dropout rate of 0.5

Dropout rate	0.0	0.2	0.5	0.8
Validation accuracy	84.2%	84.2%	84.5%	82.4%


Xception v2, different dropout rates (train)
Accuracy
Epoch
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
5
10
15
20
25
30
0.0
0.2
0.5
0.8

"Complete Exercise "

Data augmentation

- The process of generating more data from an existing dataset is called data augmentation

Data augmentation

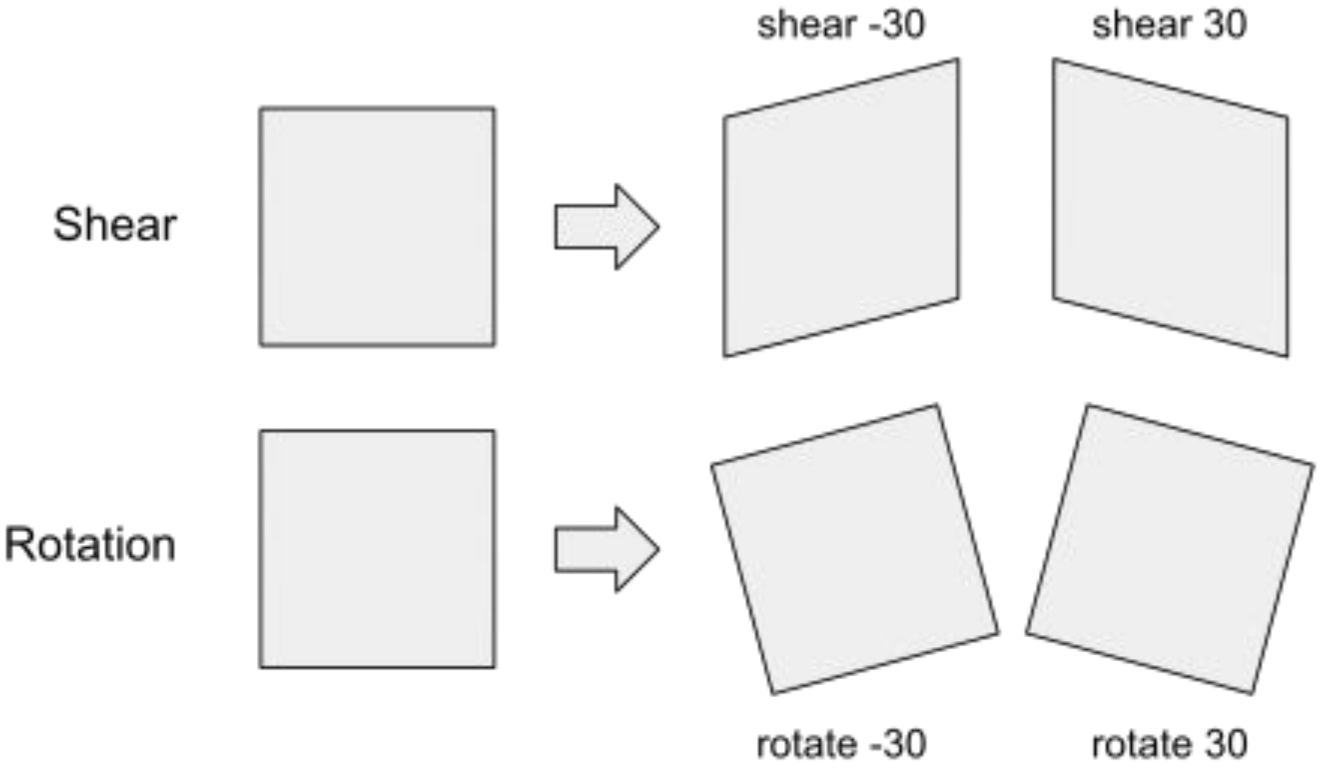
- The easiest way to create a new image from an existing one is to flip it horizontally, vertically, or both

Data augmentation

- Rotating is another image manipulation strategy that we can use: we can generate a new image by rotating an existing one by some degree

Data augmentation

- When the shear is positive, we pull the right side down, and when it’s negative, we pull the right side up



Shear
shear -30
shear 30
Rotation
rotate -30
rotate 30

Shifting an image horizontally. Positive values shift the image to the left, while negative values shift it to the right

Shifting an image vertically. Positive values shift the image to the top, while negative values shift it to the bottom

Data augmentation

- Finally, we can zoom an image in or out

Data augmentation

Data augmentation

- For example, we can create a new generator:

```
train_gen = ImageDataGenerator(
    rotation_range=30,
    width_shift_range=30.0,
    height_shift_range=30.0,
    shear_range=10.0,
    zoom_range=0.2,
    horizontal_flip=True,
    vertical_flip=False,
    preprocessing_function=preprocess_input
)
```

Data augmentation

- For our project, we'll take a small set of these augmentations:

```
train_gen = ImageDataGenerator(
    shear_range=10.0,
    zoom_range=0.1,
    horizontal_flip=True,
    preprocessing_function=preprocess_input,
)
```

Data augmentation

- We use the generator in the same way as previously:

```
train_ds = train_gen.flow_from_directory(
    "clothing-dataset-small/train",
    target_size=(150, 150),
    batch_size=32,
)
```

Data augmentation

- We load the validation dataset using exactly the same code as before:

```
validation_gen = ImageDataGenerator(
    preprocessing_function=preprocess_input
)

val_ds = validation_gen.flow_from_directory(
    "clothing-dataset-small/validation",
    target_size=image_size,
    batch_size=batch_size,
)
```

Data augmentation

- We're ready to train a new model now:

```
model = make_model(learning_rate=0.001,
droprate=0.2)
model.fit(train_ds, epochs=50,
validation_data=val_ds)
```

"Complete Exercise "

Training a larger model

- Now we're ready to train a model!

```
model = make_model(learning_rate=0.001,
droprate=0.2)
model.fit(train_ds, epochs=20,
validation_data=val_ds)
```

Using the model(Loading the model)

alexeygrigorev released this 13 hours ago · 3 commits to master since this release

Pre-trained models for chapter 7 - detecting types of clothes

Assets 4

xception_v3_44_0.853.h5	82.2 MB
xception_v4_large_08_0.894.h5	82.2 MB
Source code (zip)	
Source code (tar.gz)	

Loading the model

- To use it, load the model using the load_model function from the models package:

```
model =
keras.models.load_model('xception_v4_large_08_0.8
94.h5')
```

Evaluating the model

- To load the test data we follow the same approach: we use ImageDataGenerator, but point to the “test” directory, Let’s do it:

```
test_gen = ImageDataGenerator(
    preprocessing_function=preprocess_input
)

test_ds = test_gen.flow_from_directory(
    "clothing-dataset-small/test",
    shuffle=False,
    target_size=(299, 299),
    batch_size=32,
)
```

EVALUATING THE MODEL

- Evaluating a model in Keras is as simple as invoking the evaluate method:

```
model.evaluate(test_ds)
```

- It applies the model to all the data in the test folder and shows the evaluation metrics: loss and accuracy.

```
12/12 [==============================] - 70s
6s/step - loss: 0.2493 - accuracy: 0.9032
```

Evaluating the Model

- If we repeat the same process for the small dataset, we’ll see that the performance is worse:

```
12/12 [==============================] - 15s
1s/step - loss: 0.6931 - accuracy: 0.8199
```

Getting the predictions

- If we want to apply the model to a single image, we need to do the same thing ImageDataGenerator perform internally:

1. load an image
2. pre-process it

- We already know how to load an image. We can use load_img for that:

```
path = 'clothing-dataset-small/test/pants/c8d21106-bbdb-4e8d-83e4-bf3d14e54c16.jpg'
img = load_img(path, target_size=(299, 299))
```

```
img = load_img(path, target_size=(299, 299))
img
```

GETTING THE PREDICTIONS

- Next, we pre-process the image:

```
x = np.array(img)
X = np.array([x])
X = preprocess_input(X)
```

- And, finally, get the predictions:

```
pred = model.predict(X)
```

Getting the predictions

- We can see the predictions for the image by checking the first row of predictions: pred[0]

```
pred = model.predict(X)
pred[0]
```

```
array([-2.8609195, -4.234049 , -1.573255 , -1.9078847, 10.24705  ,
       -2.2489128, -4.297381 ,  4.43905  , -4.458805 , -3.9616926],
      dtype=float32)
```

Getting the predictions

- To get the element with the highest score, we can use the argmax method.
- It returns the index of the element with the highest score

```
pred[0].argmax()
```

```
4
```

Getting the predictions

```
labels = {
    0: 'dress',
    1: 'hat',
    2: 'longsleeve',
    3: 'outwear',
    4: 'pants',
    5: 'shirt',
    6: 'shoes',
    7: 'shorts',
    8: 'skirt',
    9: 't-shirt'
}
```

Getting the predictions

- To get the label, simply look it up in the dictionary:

```
labels[pred[0].argmax()]
```

"Complete Exercises & Lab"

Summary

- TensorFlow is a framework for building and using neural networks. Keras is a library on top of TensorFlow that makes training models simpler.

- For image processing, we need a special kind of neural networks: convolutional neural networks.

- They consist of a series of convolutional layers followed by a series of dense layers.