Machine learning - random

Aheer Srabon

1 Intro to Deep Learning

1.1 A single neuron

The linear unit works as follows in Keras,

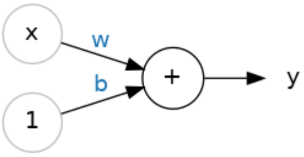


Figure 1: The linear unit: $y = wx + b$.

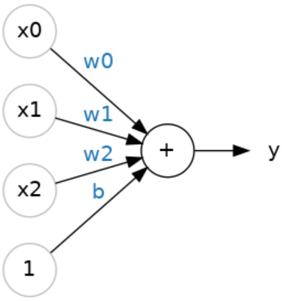


Figure 2: A linear unit with three inputs - $y = w_0x_0 + w_1x_1 + w_2x_2 + b$

```
from tensorflow import keras
from tensorflow.keras import layers

# Create a network with 1 linear unit
model = keras.Sequential([
          layers.Dense(units=1, input_shape=[3])
])
```

keras.Sequential creates a neural network as a stack of layers. The above example defines a model accepting three input features and producing a single output. The first argument units define how many outputs we want. The second argument input_shape tells keras the dimensions of the input. input_shape is the number of columns of the DataFrame except for the output column. It is a python list to permit use of more complex datasets.
Neural networks typically organize their neurons into layers. When we collect together linear units having a common set of inputs, we get a **dense** layer.

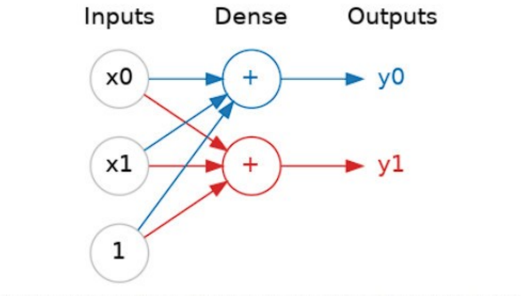


Figure 3: A dense layer of two linear units receiving two inputs and a bais.

Each layer in neural network performs some kind of relatively simple transformation. Through a deep stack of layers, a neural network can transform its inputs in more and more complex ways. In a well-trained neural network, each layer is a transformation getting us a little bit closer to a solution. A "layer" in Keras is a very general kind of thing. A layer can be, essentially, any kind of data transformation (like convolution, recurrence).

1.2 The activation function

It turns out that two dense layers with nothing in between are no better than a single dense layer by itself. *Dense layers by themselves can never move us out of the world of lines and planes.* An activation function brings non-linearity to the model. An **activation function** is simply some function that is applied to each of a layer's outputs (its activations). The most common is the rectifier function $max(0, x)$. When a attach a rectifier to a linear unit, we get a rectified linear unit or **ReLU**. Applying a ReLU activation to a linear unit means the output becomes $max(0, wx + b)$, which might be drawn like:
Stacking dense layers,
The final (output) layer is a linear unit (with no activation function). That makes this network appropriate to a *regression*, task. Other tasks (like classification) might require an activation function on the output.

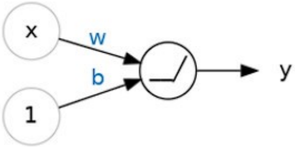


Figure 4: A rectified linear unit.

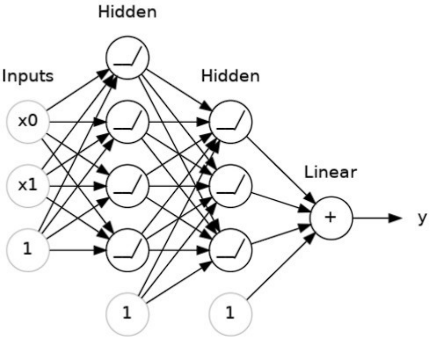


Figure 5: A stack of dense layers makes a ”fully-connected” network.

1.3 Building sequential models

The Sequential model we’ve been using will connect together a list of layers in order from first to last: the first layer gets the input, the last layer produces the output. This creates the model in the figure above:

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    # the hidden ReLU layers
    layers.Dense(units=4, activation='relu', input_shape=[2]),
    layers.Dense(units=3, activation='relu'),
    # the linear output layer
    layers.Dense(units=1),
])
```

Sequential takes a list of layers. Sometimes, some other layer are put between the Dense layer and its activation function. In these cases, the activation can be defined on its own Activation layer.

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    # the hidden ReLU layers
    layers.Dense(units=4, activation='relu', input_shape=[2]),
    layers.Dense(units=3)
    # The activation is in its own layer. Between
    # the Dense layer and the Activation layer, some other layers can be
    # put
    layers.Activation('relu')
    # the linear output layer
    layers.Dense(units=1),
])
```

To train a neural network, the following things are needed,

- The model iteself
- Training data
- A loss function (tells the network what problem to solve)
- An optimizer (tells the network how to solve the problem)

Some example of loss functions are,

- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)
- Huber loss

1.4 Stochastic gradient descent

Virtually all of the optimization algorithms used in deep learning belong to a family called **stochastic gradient descent**. They are iterative algorithms that train a network in steps.

1. Sample some training data and run it through the network to make predictions
2. Measure the loss between the predictions and the true values
3. Finally, adjust the weights in a direction that makes the loss smaller

Repeat the above process until the loss as small as necessary (or until it won't decrease any further). Each iteration's sample of training data is called a **minibatch** (or often just "batch"), while a complete round of the training data is called an **epoch**. A smaller learning rate means the network needs to see more **minibatches** before its weights converge to their best value. The number of epochs you train for is how many times the network will see each training example. *The learning rate and the size of the minibatches are the two parameters that have the largest effect on how the SGD training proceeds.*
Stochastic means "determined by chance". The training is stochastic because the minibatches are random samples from the dataset.
A common loss function for regression problems is **mean absolute error** or **MAE**.
The loss function tells the network its objective. The optimizer is an algorithm that adjusts the weights of the model to minimize the loss. The learning rate and the size of the minibatches are the two parameters that have the largest effect on how the SGD training proceeds.
Adam is an SGD algorithm that has an adaptive learning rate that makes it suitable for most problems without any parameter tuning. Adam is a great general purpose optimizer. After defining a model, a loss function and optimzer with the models compile method can be defined,

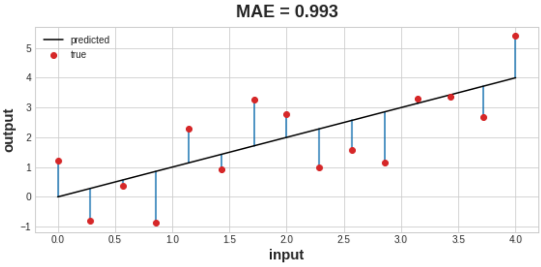


Figure 6: The mean absolute error is the average length between the fitted curve and the data points.

```
model.compile(
          optimzer="adam",
          loss="mae",
)
```

Splitting the data into training data and validation data,

```
import pandas as pd
from IPython.display import display

red_wine = pd.read_csv('../input/dl-course-data/red-wine.csv')

# Create training and validation splits
df_train = red_wine.sample(frac=0.7, random_state=0)
df_valid = red_wine.drop(df_train.index)
display(df_train.head(4))

# Scale to [0, 1]
max_ = df_train.max(axis=0)
min_ = df_train.min(axis=0)
df_train = (df_train - min_) / (max_ - min_)
df_valid = (df_valid - min_) / (max_ - min_)

# Split features and target
X_train = df_train.drop('quality', axis=1)
X_valid = df_valid.drop('quality', axis=1)
y_train = df_train['quality']
y_valid = df_valid['quality']

_, input_shape = X_train.shape
```

Choose a three-layer network with over 1500 neurons.

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
     layers.Dense(512, activation='relu', input_shape=[11]),
     layers.Dense(512, activation='relu'),
     layers.Dense(512, activation='relu'),
     layers.Dense(1),
])
```

Deciding the architecture of the model should be part of the process. Start simple and use the validation loss as guide. Compile the optimzer and loss function,

```
model.compile(
        optimizer='adam',
        loss='mae'
)
```

The following code tells Keras to feed the optimizer 256 rows of the training data at a time (the batch_size) and to do that 10 times all the way through the dataset (the epochs),

```
history = model.fit(
        X_train, y_train,
        validation_data = (X_valid, y_valid),
        batch_size=256,
        epochs=10,
)

# convert the training history to a dataframe
history_df = pd.DataFrame(history.history)
history_df['loss'].plot()
```

1.5 Overfitting and Underfitting

Information of the training data as being two kinds,

- *Signal* - is the part that generalizes, the part that can help the model to make predictions from new data. Useful.
- *Noise* - is the part that is only true for the training data. It is all of the random fluctuation that comes from data in the real-world or all of the incidental, non-informative patterns that can't actually help the model make predictions. Not useful.

The training loss will go down either when the model learns signal or when it learns noise. But the validation loss will go down only when the model learns signal (whatever noise the model learned from the training set won't generalize to new data). So, when a model learns signal, both curves go down, but when it learns noise, a gap is created in the curves. The size of the gap tells us how much noise the model has learned.
Underfitting vs overfitting,

- **Underfitting** the training set is when the loss is not as low as it could be because the model hasn't learned enough *signal.*
- **Overfitting** the training set is when the loss is not as low as it could be because the model learned too much *noise.*

A model's *capacity* refers to the **size** and **capacity** of the patterns it is able to learn,

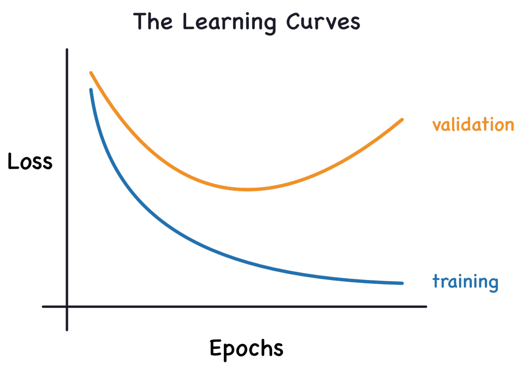


Figure 7: The validation loss gives an estimate of the expected error on unseen data.

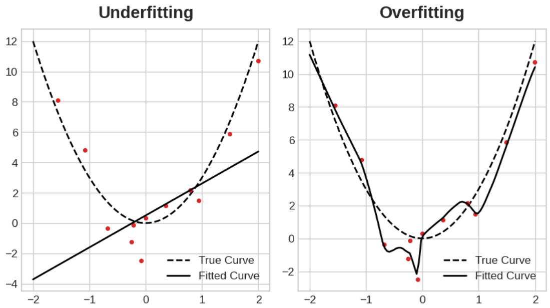


Figure 8: Underfitting vs overfitting

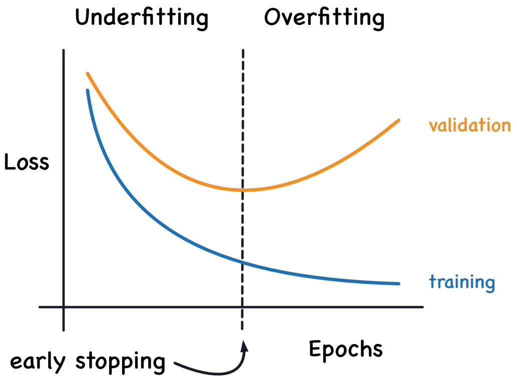


Figure 9: We keep the model where the validation loss is at a minimum - early stopping

- Wider neural network – more units to existing layers. Learns linear relationships easily
- Deeper neural network – adding more layers. Learns non-linear relationships easily

Once we detect that the validation loss is starting to rise again, we can reset the weights back to where the minimum occured. This is called *early stopping.* In Keras, early stopping can be included through a callback. A callback is a function that is run every so often while the network trains. The early stopping callback will run after every epoch.

```
from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(
    min_delta=0.001, # minimium amount of change to count as an improvement
    patience=20, # how many epochs to wait before stopping
    restore_best_weights=True,
)
```

These parameters say: ”If there hasn’t been at least an improvement of 0.001 in the validation loss over the previous 20 epochs, then stop the training and keep the best model you found.”

Train a neural network,

```
import pandas as pd
from tensorflow import keras
from tensorflow.keras import layers, callbacks

red_wine = pd.read_csv('../input/dl-course-data/red-wine.csv')

# Create training and validation splits
df_train = red_wine.sample(frac=0.7, random_state=0)
df_valid = red_wine.drop(df_train.index)

# Scale to [0, 1]
max_ = df_train.max(axis=0)
min_ = df_train.min(axis=0)
df_train = (df_train - min_) / (max_ - min_)
df_valid = (df_valid - min_) / (max_ - min_)

# Split features and target
X_train = df_train.drop('quality', axis=1)
X_valid = df_valid.drop('quality', axis=1)
y_train = df_train['quality']
y_valid = df_valid['quality']


early_stopping = callbacks.EarlyStopping(
    min_delta=0.001, # minimium amount of change to count as an improvement
    patience=20, # how many epochs to wait before stopping
    restore_best_weights=True,
)
```

```
model = keras.Sequential([
    layers.Dense(512, activation='relu', input_shape=[11]),
    layers.Dense(512, activation='relu'),
    layers.Dense(512, activation='relu'),
    layers.Dense(1),
])
model.compile(
    optimizer='adam',
    loss='mae',
)
history = model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=256,
    epochs=500,
    callbacks=[early_stopping], # put your callbacks in a list
    verbose=0,  # turn off training log
)

history_df = pd.DataFrame(history.history)
history_df.loc[:, ['loss', 'val_loss']].plot();
print("Minimum validation loss: {}".format(history_df['val_loss'].min()))
```

1.6 Dropout and batch normalization

Dropout can help correct overfitting. Overfitting is caused by the network learning spurious patterns in the training data. To recognize these spurious patterns, a network will often rely on a very specific combinations of weight, a kind of ”conspiracy” of weights. Being so specific, they tend to be fragile: remove one and the conspiracy falls apart. This is the idea behind **dropout** – to break up these conspiracies. Some fraction of a layer's input units is dropped in every step of training, making it much harder to learn those spurious patterns in the training data. Instead, it has to search for broad, general patterns, whose weight patterns tend to be more robust. Dropout can also be thought of as being an ensemble of networks. The predictions will no longer be made by one big network, but instead by a committe of smaller networks.

```
keras.Sequential([
    # ...
    layers.Dropout(rate=0.3), # apply 30% dropout to the next layer
    layers.Dense(16),
    # ...
])
```

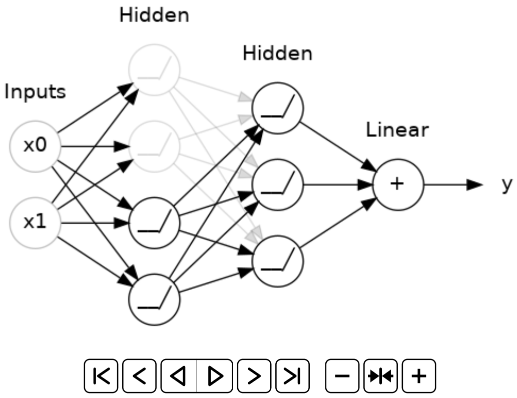


Figure 10: Here, 50% dropout has been added between the two hidden layers.

Batch normalization can help correct training that is slow or unstable. It is a good idea to put all of the data on a common scale because SGD will shift the network weights in proportion to how large an activation the data produces. Features that tend to produce activations of very different sizes can make for unstable training behavior. **Batch normalization layer** normalizes inside the network. A batch normalization layer looks at each batch as it comes in, first normalizing the batch with its own mean and standard deviation, and then also putting the data on a new scale with two trainable rescaling parameters. Batchnorm, in effect, performs a kind of co-ordinated rescaling of its inputs.

- Models with batchnorm tend to need fewer epochs to complete training
- Batchnorm is added as an aid to the optimizaiton process (though it can sometimes also help prediction performance)
- Batchnorm can fix various problems that can cause the training to get stuck

```
# after a layer
layers.Dense(16, activation='relu'),
layers.BatchNormalization(),

# between a layer and its activation function
layers.Dense(16),
layers.BatchNormalization(),
layers.Activation('relu'),
```

Using dropout and batch normalization,

```
# Setup plotting
import matplotlib.pyplot as plt

from tensorflow import keras
from tensorflow.keras import layers

plt.style.use('seaborn-whitegrid')
# Set Matplotlib defaults
```

```
plt.rc('figure', autolayout=True)
plt.rc('axes', labelweight='bold', labelsize='large',
       titleweight='bold', titlesize=18, titlepad=10)


import pandas as pd
red_wine = pd.read_csv('../input/dl-course-data/red-wine.csv')

# Create training and validation splits
df_train = red_wine.sample(frac=0.7, random_state=0)
df_valid = red_wine.drop(df_train.index)

# Split features and target
X_train = df_train.drop('quality', axis=1)
X_valid = df_valid.drop('quality', axis=1)
y_train = df_train['quality']
y_valid = df_valid['quality']

model = keras.Sequential([
    layers.Dense(1024, activation='relu', input_shape=[11]),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(1),
])
model.compile(
    optimizer='adam',
    loss='mae',
)

history = model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=256,
    epochs=100,
    verbose=0,
)


# Show the learning curves
history_df = pd.DataFrame(history.history)
history_df.loc[:, ['loss', 'val_loss']].plot()
```

1.7 Binary classification

The main difference is in the loss function we use and in what kind of outputs we want the final layer to produce.

Accuracy is one of the many metrics in use for measureing success on a classification problem. Accuracy is the ratio of correct predictions to total predictions,

$$accuracy = \frac{number\_correct}{total} \tag{1}$$

The problem with accuracy (and most other classification metrics) is that it can't be used as a loss function. SGD needs a loss function that changes smoothly, but accuracy, being a ratio of counts, changes in "jumps". So, a substitute is chosen to act as the loss function. This substitute is the *cross-entropy* function.

The loss function defines the objective of the network during training. With regression, the goal was to minimize the distance between the expected outcome and the predicted outcome. MAE was chosen to measure this distance. For classification, what is wanted instead is a distance between *probabilities*, and this is what cross-entropy provides. **Cross-entropy** is a sort of measure for the distance from one probability distribution to another. The idea is that we want our network to predict the correct class with probability 1.0. The further away the predicted probability is from 1.0, the greater will be the cross-entropy loss.

Binary classification,

```
import pandas as pd
from tensorflow import keras
from tensorflow.keras import layers

ion = pd.read_csv('../input/dl-course-data/ion.csv', index_col=0)

df = ion.copy()
df['Class'] = df['Class'].map({'good': 0, 'bad': 1})

df_train = df.sample(frac=0.7, random_state=0)
df_valid = df.drop(df_train.index)

max_ = df_train.max(axis=0)
min_ = df_train.min(axis=0)

df_train = (df_train - min_) / (max_ - min_)
df_valid = (df_valid - min_) / (max_ - min_)
df_train.dropna(axis=1, inplace=True) # drop the empty feature in column 2
df_valid.dropna(axis=1, inplace=True)

X_train = df_train.drop('Class', axis=1)
X_valid = df_valid.drop('Class', axis=1)
y_train = df_train['Class']
y_valid = df_valid['Class']

model = keras.Sequential([
    layers.Dense(4, activation='relu', input_shape=[33]),
    layers.Dense(4, activation='relu'),
    layers.Dense(1, activation='sigmoid'),
])
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['binary_accuracy'],
)
early_stopping = keras.callbacks.EarlyStopping(
    patience=10,
    min_delta=0.001,
    restore_best_weights=True,
)

history = model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=512,
    epochs=1000,
    callbacks=[early_stopping],
    verbose=0, # hide the output because we have so many epochs
)

history_df = pd.DataFrame(history.history)
# Start the plot at epoch 5
history_df.loc[5:, ['loss', 'val_loss']].plot()
history_df.loc[5:, ['binary_accuracy', 'val_binary_accuracy']].plot()

print(("Best Validation Loss: {:0.4f}" +\
      "\nBest Validation Accuracy: {:0.4f}")\
      .format(history_df['val_loss'].min(),
              history_df['val_binary_accuracy'].max()))
```

2 Computer vision

2.1 Convolution

$$(f \star g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau \tag{2}$$

Convolution is the integral of the product of the two functions after one is reflected about the y axis and shifted. The integral is evaluated for all values of shift, producing the convolution function. It expresses how the shape of one function is modified by the other. At each t the convolution formula can be described as area under the function $f(\tau)$ weighted by the function $g(-\tau)$ shifted by amount t.

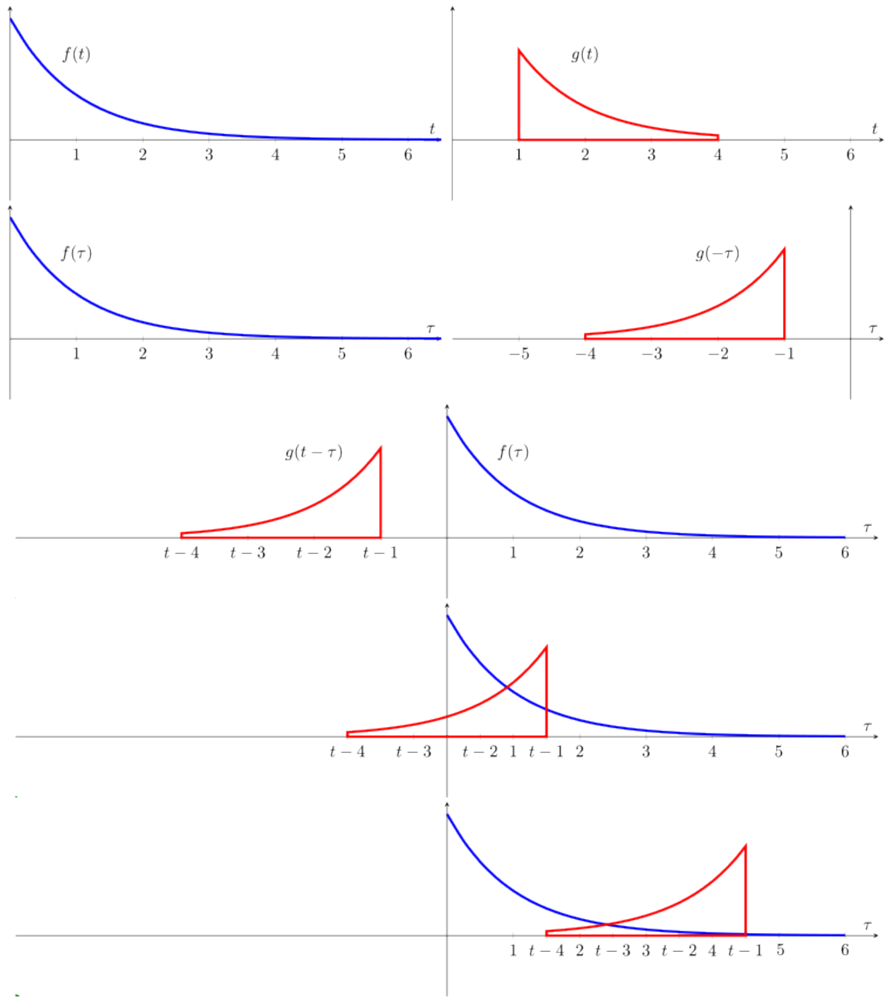


Figure 11:

- Express each function in terms of a dummy variable τ.

- Reflect one of the functions: $g(\tau) \to g(-\tau)$.
- Add a time-offset t, which allows $g(-\tau)$ to slide along the τ-axis. If t is a positive value, then $g(t-\tau)$ is equal to $g(-\tau)$ that slides or is shifted along the τ-axis toward the right (toward $+\infty$) by the amount of t. If t is a negative value, then $g(t-\tau)$ is equal to $g(-\tau)$ that slides or is shifted towards the left (towards $-\infty$) by the amount of $|t|$.
- Start t at $-\infty$ and slide it all the way to $+\infty$. Wherever the two functions intersect, find the integral of their product. In other words, at time t, compute the area under the function $f(\tau)$ weighted by the weighting function $g(t-\tau)$.

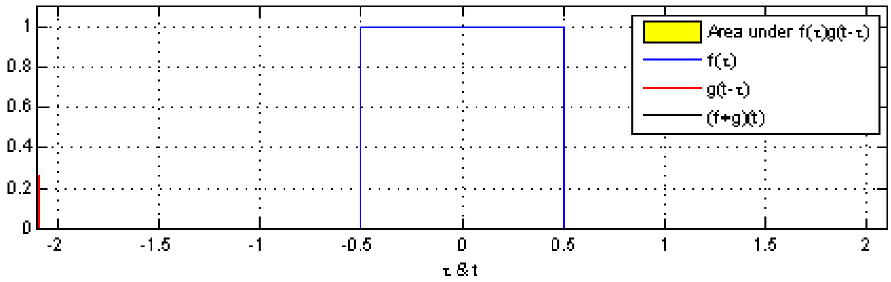


Figure 12: Convolution of box signal with itself

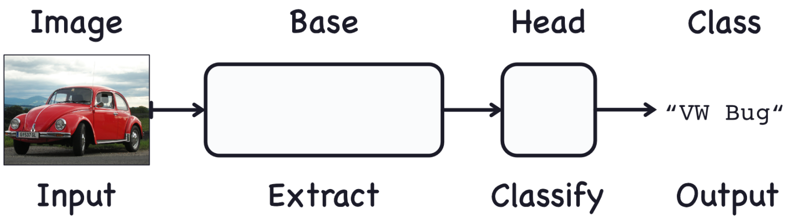
2.2 The convolutional classifier

CNN are great at computer vision. CNN are also used in,

- Generative Adversarial Networks (GAN)
- Image segmentation

A convnet used for image classification has two parts,

- *Convolutional base* - is used to extract the features from an image. The convolutional base is formed primarily of layers performing the convolution operation.
- *Dense head* - is used to **determine the class** of the image. The dense head is formed primarily of dense layers.



photograph by Dnalor 1 (Wikimedia Commons) CC-BY-SA 3.0

Figure 13: The convolutional classifier

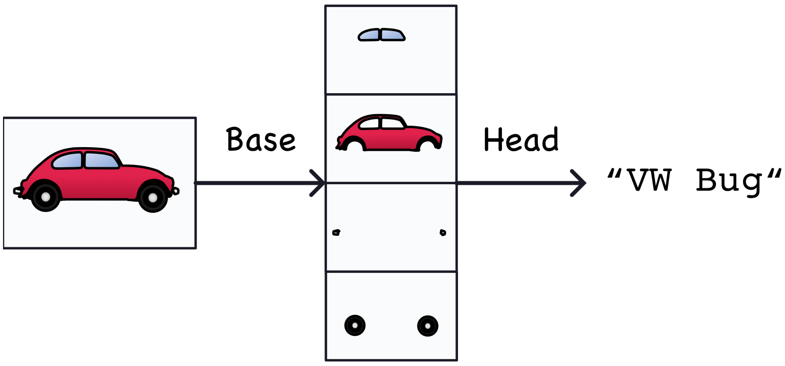


Figure 14: Feature extraction and classification

The goal of the network during training is to learn two things,

- which features to extract (perfomed in base)
- which class goes with what features (perfomed in head)

Most often, the base of pretrained models is reused. To the pretrained base, an untrained head is attached. So before training, the CNN can already extract features from images. When an untrained head is attached to it and trained, it can learn to perform the current task. Because the head usually consists of only a few dense layers, very accurate classifiers can be created from relatively little data.

```
# Imports
import os, warnings
import matplotlib.pyplot as plt
from matplotlib import gridspec
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing import image_dataset_from_directory
import pandas as pd

'''Load the data'''

# Reproducability
def set_seed(seed=31415):
    np.random.seed(seed)
    tf.random.set_seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    os.environ['TF_DETERMINISTIC_OPS'] = '1'
set_seed(31415)

# Set Matplotlib defaults
plt.rc('figure', autolayout=True)
plt.rc('axes', labelweight='bold', labelsize='large',
       titleweight='bold', titlesize=18, titlepad=10)
plt.rc('image', cmap='magma')
warnings.filterwarnings("ignore") # to clean up output cells


# Load training and validation sets
ds_train_ = image_dataset_from_directory(
    '../input/car-or-truck/train',
    labels='inferred',
    label_mode='binary',
    image_size=[128, 128],
    interpolation='nearest',
    batch_size=64,
    shuffle=True,
)
ds_valid_ = image_dataset_from_directory(
    '../input/car-or-truck/valid',
    labels='inferred',
    label_mode='binary',
    image_size=[128, 128],
    interpolation='nearest',
    batch_size=64,
    shuffle=False,
)

# Data Pipeline
def convert_to_float(image, label):
    image = tf.image.convert_image_dtype(image, dtype=tf.float32)
    return image, label

AUTOTUNE = tf.data.experimental.AUTOTUNE
ds_train = (
    ds_train_
    .map(convert_to_float)
    .cache()
    .prefetch(buffer_size=AUTOTUNE)
)
ds_valid = (
    ds_valid_
    .map(convert_to_float)
    .cache()
    .prefetch(buffer_size=AUTOTUNE)
)

'''Define pretrained base'''
pretrained_base = tf.keras.models.load_model(
    '../input/cv-course-models/cv-course-models/vgg16-pretrained-base',
)
pretrained_base.trainable = False
```

```
'''Attach head'''
model = keras.Sequential([
    pretrained_base,
    layers.Flatten(),
    layers.Dense(6, activation='relu'),
    layers.Dense(1, activation='sigmoid'),
])

'''Train'''
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['binary_accuracy'],
)

history = model.fit(
    ds_train,
    validation_data=ds_valid,
    epochs=30,
    verbose=0,
)

history_frame = pd.DataFrame(history.history)
history_frame.loc[:, ['loss', 'val_loss']].plot()
history_frame.loc[:, ['binary_accuracy', 'val_binary_accuracy']].plot();
```

2.3 Convolution and ReLU

Two most important types of layers that can be found in the **base** of a convolutional image classifier,

- convolutional layer with ReLU activation
- Maximum pooling layer

As the activation layers can be a separate layer, *convolution*, *ReLU*, and *maximum pooling* are used to extract features in the base.
The feature extraction performed by the base has three operations,

- *Filter* - an image for a particular feature (convolution)
- *Detect* - the feature within the filtered image
- *Condense* - the image to enhance the feature

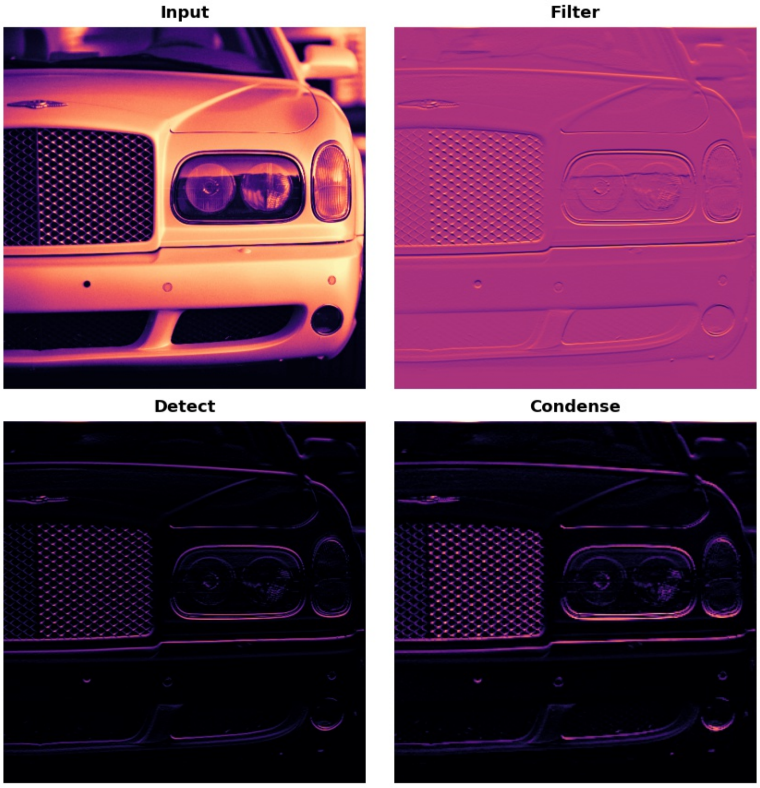


Figure 15: Feature extraction and classification