Chapter 1 – Introduction to Machine Learning

Chapter 1.1 – ML Code

The Syntax

1 – Reading the Data

```
# For the updated Python, use .values() instead of
as_matrix() to do things.

df = pd.read_csv("data.csv")
df.head()
df.info()

X_features = df[[0,1,2,3,4]].values
Y_target = df[5].values
```

2 – Preparing the Model

```
# The syntax of ML codes usually follows below. You
call the model to instantiate it by ModelName(). Then
you train the model with fit(). In fact, "fit" is the
same word as "train"! THEY ARE INTERCHANGEABLE WORDS
IN SCIKITLEARN. You then make predictions with
predict(). Then you assess its accuracy /
classification rate with score(); it returns accuracy
with classifier models and R^2 with regression models.

import smth from somewhere

model = RandomForestClassifier()
model.fit(X, Y)
predictions = model.predict(X)
model.score(X, Y)
```

3 – Train Test Split

```
# You also want train-test sets for predictions.

from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test =
train_test_split(data.data, data.target,
test_size=0.33)
```

4 – Normalise / Scale the Data

```
# You may ask, what is the difference between
fit_transform() and transform()? Why must use the
former? It's because StandardScaler works like a model.
You need to fit() it first, before you transform. It
allows the function to learn about the data you have.

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
x_train2 = scaler.fit_transform(x_train)
x_test2 = scaler.transform(x_test)
```

Chapter 2: Introduction to Tensorflow 2.0

Chapter 2.1: Setting Up TF2 in Google Colab

Setting Up the Environment

```
%tensorflow_version 2.x

import tensorflow as tf
print(tf.__version__)
```

Importing Data into Google Colab

a) Downloading Data from a URL

```
# To do this, we can use !wget <url> to download the
file and it'll be stored in our current directory. You
can check what is in our cd by typing !ls. You can
also check if the file has a header row by typing
!head <document>.

!wget https://....
!ls
!head arrhythmia.data

import pandas as pd
df = pd.read_csv('arrhythmia.data', header=None)
data = df[[0,1,2,3,4,5]]
data.columns = ['age', 'sex', 'height', 'weight', 'QRS
duration', 'P-R interval']
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [15, 15] # make the
plot bigger so the subplots don't overlap
data.hist(); # semicolon to suppress return value

from pandas.plotting import scatter_matrix
scatter_matrix(data);
```

b) Using tf.keras

```
# Here we want to use get_file() to download. We
first save the url into a variable.

url = 'blahblah'
tf.keras.utils.get_file('auto-mpg.data', url)
!head /root/.keras/datasets/auto-mpg.data

df = pd.read_csv('data', header=None,
delim_whitespace=True)
df.head()
```

c) Upload Directly

```
from google.colab import files
uploaded = files.upload()
```

d) Access from Google Drive

```
from google.colab import drive
drive.mount('/content/gdrive')
```

Chapter 2.2: TF2 and Keras

Understanding Keras

Instantiate the Model

```
# When we implement TF2 with Keras, we need to mention
the Input() to tell keras the size of the input vector
(no. of features). We use Dense() to specify the
output size (which is 1 below) and the activation
function. If you had noticed, we put both Input() and
Dense() objects into a list, and then passed it into
Sequential().

model = tf.keras.models.Sequential([
       tf.keras.layers.Input(shape=(D,)),
       tf.keras.layers.Dense(1, activation='sigmoid')
])
```

Fit / Train the Model

```
# After doing the above, we need tune our model to
tell it how we want it to run. We will start with a
cost / loss function; the less error then the closer
it is to 0. The optimizer is the method of doing
gradient descent; there are many ways, and we choose
adam here. For metrics, we choose accuracy because we
are doing categorical here.

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

# We then fit the model here with the actual data. We
can then assess the loss and validation loss to
determine if we need more or less epochs – the curve
should taper off at the end. If tail is too long,
means too many epochs; if still going down, too few.

r = model.fit(x_train, y_train,
              validation_data = (x_test, y_test),
              epochs = 100)

plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
```

CLASSIFICATION

 # Install TF2

```
%tensorflow_version 2.x

import tensorflow as tf
print(tf.__version__)
```

 # Dataset

```
from sklearn.datasets import load_breast_cancer

data = load_breast_cancer()
print(type(data)) # check datatype. It is a Bunch
object, akin to a dictionary where you can treat the
keys like attributes.

print(data.keys()) # see what keys you have
print(data.data.shape) # 569 samples, 30 features

print(data.target) # 1D array of targets
print(data.target_names) # to know what target means
print(data.target.shape) # 569 rows: y same len as x.
print(data.feature_names)
```

 # Train Test Split

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test =
train_test_split(data.data, data.target, test_size =
0.33)

print(x_train.shape) # 381 rows 30 features
N, D = x_train.shape
```

 # Scale the Data

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)
```

 # Modelling

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Input(shape=(D,)),
  tf.keras.layers.Dense(1, activation = 'sigmoid')
])

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

r = model.fit(x_train, y_train, validation_data =
(x_test, y_test), epochs = 100)

print("Train score:", model.evaluate(x_train,
y_train))
print("Test score:", model.evaluate(x_test, y_test))
```

 # Plotting the Loss

```
import matplotlib.pyplot as plt

plt.plot(r.history['loss'], label = 'loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
```

 # Plotting the Accuracy

```
plt.plot(r.history['accuracy'], label = 'acc')
plt.plot(r.history['val_accuracy'], label = 'val_acc')
plt.legend()
```

 # Predictions

```
P = model.predict(x_test)
print(P) # outputs of sigmoid, probabilities P(Y=1|x)

P = np.round(P).flatten() # round the predictions +
flatten as targets size (N,) but predictions (N,1)
print(P)

print("Manually calculated accuracy:", np.mean(P == y_test))
print("Evaluate output:", model.evaluate(x_test, y_test))
```

REGRESSION

 # Install TF2 & Other Libraries

```
%tensorflow_version 2.x

import tensorflow as tf
print(tf.__version__)

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

 # Data

```
!wget https://raw.githubusercontent.com/
lazyprogrammer/machine_learning_examples/
master/tf2.0/moore.csv

data = pd.read_csv('moore.csv', header=None).values
x = data[:,0].reshape(-1, 1) # make it 2D array of
size N x D. -1 means the -1th index, aka last index,
which tells it the length of the col.
y = data[:,1]
```

 # Plot Data

```
plt.scatter(x,y)
```

 # Modelling

Note how there is no need for activation function at all, because it's continuous numbers! Furthermore, we use **stochastic gradient descent** for the optimizer. Lastly, we must take note of **learning rate scheduling**, where we modify the learning rate at each epoch. SGD(learning_rate, momentum).

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Input(shape=(1,)),
  tf.keras.layers.Dense(1)
])

model.compile(optimizer=tf.keras.optimizers.SGD(0.001,
0.9), loss = 'mse')

# Learning Rate Scheduler
def schedule(epoch,lr):
  if epoch >= 50:
    return 0.0001
  return 0.001

scheduler =
tf.keras.callbacks.LearningRateScheduler(schedule)
```

 # Transform Exponential Graph into Linear

```
y = np.log(y)
plt.scatter(x,y)
```

 # Normalise the Data

```
# 1970 to 2018 are very large numbers, so we would
like them to be near zero.
# So we shall minus by their means.
# We don't divide by SD as it is not very
representative of one year.

x = x - x.mean()
```

 # Train the Model

```
r = model.fit(x, y, epochs = 200,
callbacks=[scheduler])
```

 # Plotting the Loss

```
plt.plot(r.history['loss'], label = 'loss')
```

 # Slope of the Line

```
# The slope of the line is related to the doubling
rate of transistor count.

print(model.layers) # Note: there is only 1 layer, the
"Input" layer doesn't count
print(model.layers[0].get_weights()) # arrays of w, b.
W.shape = (D,M) where b.shape = (M,); D is input size,
M is output size.

a = model.layers[0].get_weights()[0][0,0] # slope of
the line
print("Time to double:", np.log(2) / a)
```

 # Predictions

```
yhat = model.predict(x).flatten()
plt.scatter(x,y)
plt.plot(x, yhat)

w, b = model.layers[0].get_weights()
x = x.reshape(-1,1) # reshape as we flattened it

yhat2 = (x.dot(w) + b).flatten()

np.allclose(yhat, yhat2) # floating points should not
use == because of round-off errors. Gives True.
```

Saving the Model

```
# This is only possible if you did not create a model explicitly. Have to use the below method:

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(1, input_shape = (D,), activation = ‘sigmoid’))

model.save(‘linearclassifier.h5’)

!ls -lh # to check it exists
```

Load the Model

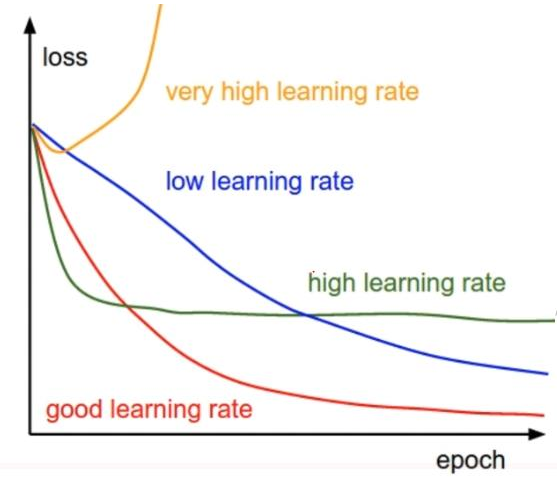
```
model = tf.keras.models.load_model(‘linearclassifier.h5’)
print(model.layers)
model.evaluate(x_test, y_test)

# OR FOR GOOGLE COLAB (only works in Chrome)

from google.colab import files
files.download(‘linearclassifier.h5’)
```

Chapter 2.3: Brief Introduction to Neural Networks

How Does the Model Learn?

- For regression, the error function is **MSE** and it is something we want to minimise. The minimum point of MSE is when the derivative is equals to 0, i.e. the slope = 0.
- When there are **multivariable**, then the equivalent of the derivative is called the **gradient**. This gradient is just a vector or tensor of partial derivatives for each of the variables we're taking the gradient with respect to. Hence to find where error is the lowest, we set the gradient = 0, and hence find the best model parameters (weights and biases).
- We can automatically calculate these weights and biases using **TF's automatic differentiation**. TF will automatically find the gradient of loss to all model weights, and it uses the gradients to train the model.
- **Q**: since we can just set gradient = 0 and solve for w and b, why do we need to do iterations / epochs? Why cannot just do once? **A:** It is only possible to do once for linear regressions. For other types, we need **gradient descent**. This happens inside Keras's fit() function: we start with a random starting point, and then iterate by taking small steps to update w and b, until we converge at a final value.
- The value of the small steps is based on **eta** that specifies the learning rate and how fast or slow we want to train our model. It is important to set the LR right or else the model will fail even if it is a good model. There is no good method to choosing the hyperparameter η and it (along with all other hyperparameters) has to be chosen by mostly trial and error or intuition. Hence, we need **learning rate scheduling**. Remember that if LR is too high, your loss will keep on increasing as it will **oscillate** or it may converge to a suboptimal value. If LR too low, then very slow. Normally we try powers of 10: 0.1, 0.01, 0.001 etc.

```
w, b = randomly initialized
for epoch in range(epochs):
    w = w - η * ∇_w J
    b = b - η * ∇_b J
```

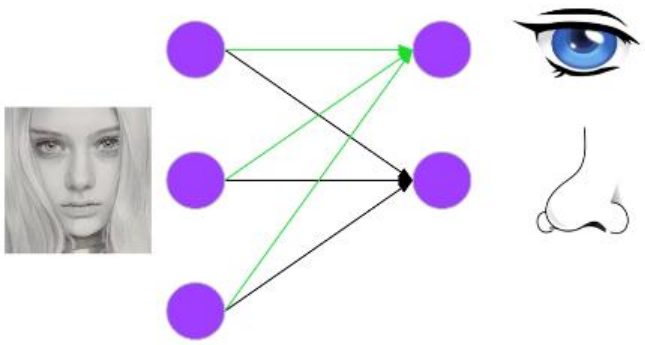


Chapter 3: Artificial Neural Networks (ANNs)

Why NN? Researchers noticed that each layer learns increasingly complex features. For example, faces in pictures get more defined. That is why we want to go deep.

Chapter 3.1: Feedforward Neural Networks

A **feedforward neural network** just means that the signal goes from left to right only.



Each neuron is calculating something different given the same inputs. For example, when looking at a face, one neuron may be looking for the presence of an eye, while another for a nose. In other words, **different neurons are looking for different features**.

We can assume that a neuron in any layer is the same type of neuron you can find in any layer. Hence, there is a uniform structure.

An NN is scalable as its **width** can be increase by increasing the number of nodes per layer, while its **length** can be increased as one layer can act as inputs to another layer.

Classification & Regression

Line: $ax + b$
Neuron: $z = \sigma(w^T x + b)$

The neuron is read as "transposed weight matrix time x, plus bias, then sigmoid it to a value between 0 or 1".

Node z is a vector of size M (column vector of size Mx1) while x is a vector of size D (column vector of size Dx1). Hence, w^T is a transposed matrix of size MxD, and b is a vector of size M. The sigmoid σ() is an element-wise operation.

If we are doing regression, then we will not want to sigmoid the last layer. Instead, do as per normal but just do not sigmoid it. It'll be wx + b still which is like a linear regression.

Note how at the second layer, the neuron will be: $w_2^T \sigma(w_1^T x + b_1) + b_2$. Also note how each neuron is **non-linear** due to the sigmoid function.

Activation Functions

1 - Sigmoid

Pros: The sigmoid is generally useful because it can map values to between 0 and 1 as a probability, and also makes the NN's decision boundary nonlinear.

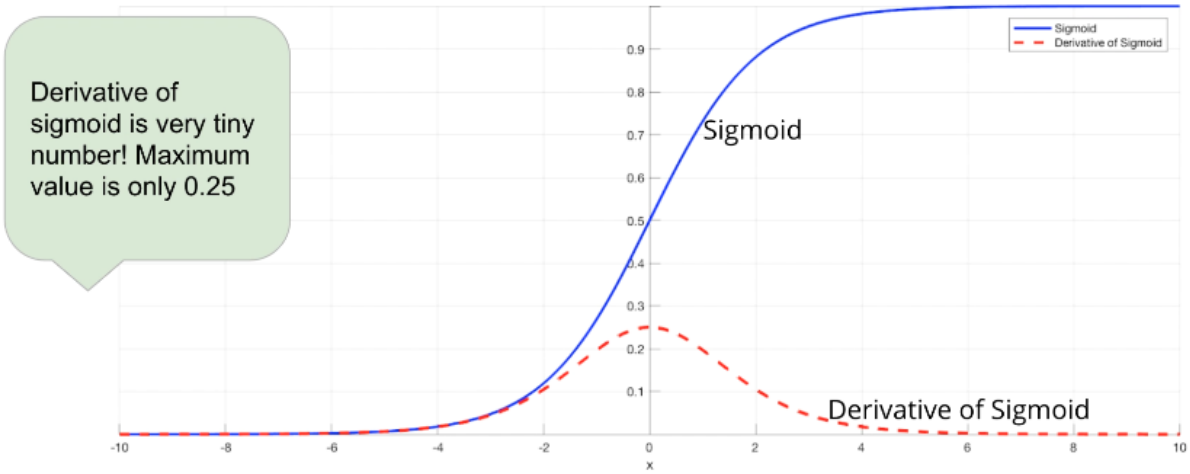
Cons: However, if we used standardised data (inputs centred around 0), sigmoid will change it to be centred around 0.5 (since 0 to 1). And yet it feeds to the next layer, creating problems.

2 – Tanh Hyperbolic Tangent

tanh(x) = sinh(x) / cosh(x) fixes this problem as it has the same shape as sigmoid, but goes from -1 to 1. It is a scaled and vertically shifted sigmoid.

However, both sigmoid and tanh are still problematic due to the **vanishing gradient problem**. Usually we use gradient descent to train a model, and because we do backwards propagation, then the deeper the NN, the more terms have to be

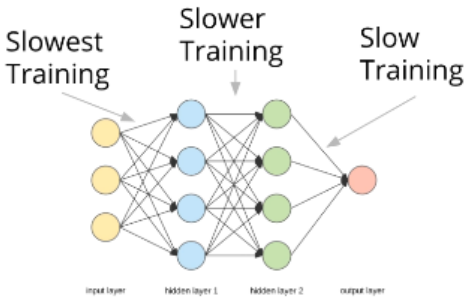
multiplied in the gradient due to the chain rule of calculus. In other words, we end up multiplying by the derivative of the sigmoid over and over again.

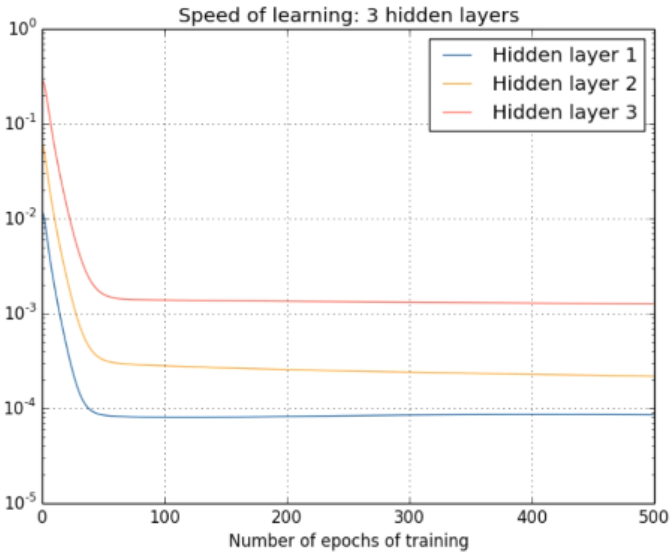


This is bad because as we multiply the derivative many times (which its maximum value is 0.25), we will get an even smaller number. This means that the further we go back in a NN, the smaller the gradient becomes. This is the vanishing gradient problem. So what about this problem then?

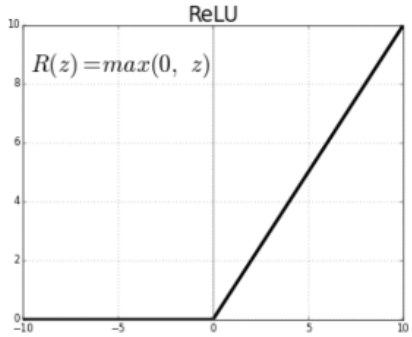
The Vanishing Gradient Problem

- Note: graph is in **log scale**
- The further back we go, the smaller the gradient!
- Modern deep networks have **hundreds** of layers



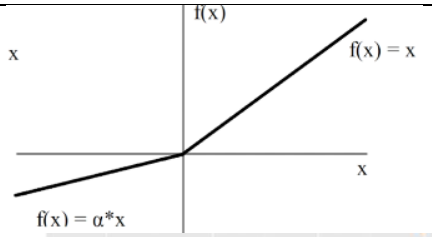


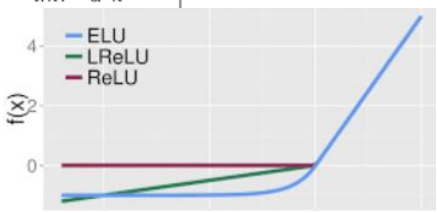
Basically, as we go as back as ever, the gradient is so small. Remember that the training algorithm is to take small steps in the direction of the gradient. And if the gradient is small / nearly zero, the weights and biases do not get updated, resulting that the earlier hidden layers not being trained at all.

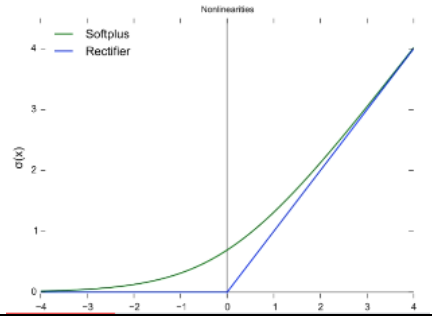


To solve this problem, we simply **do not use activation functions with vanishing gradients**, which led to the usage of **ReLU (rectifier linear unit)**.

The ReLU has no vanishing gradient because the left half is already vanished! It leads to a problem called the **"dead neuron" problem**, which are neurons that always output zero because the weighted sum of its inputs are always less than or equals to zero. However practically, this is not much of a problem at all. As long as the right side does not vanish, the model still works.







Some researchers have tried to solve it with the **Leaky ReLU LReLU**. It has a small positive slope for negative inputs so the whole thing still works like tanh. Another method is the **ELU exponential linear unit**, which allows the outputs to be negative.

Another one is **softplus**. Both ELU and softplus have vanishing gradients on the left but they are not too problematic because ReLU works.

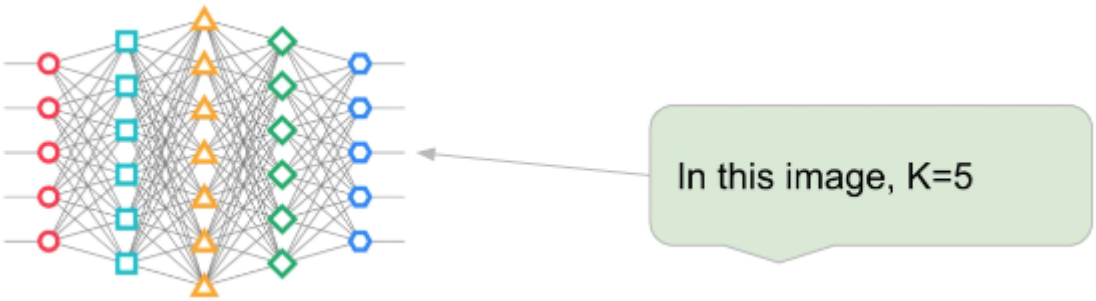
However, remember that we wanted our values to be centred around 0? ReLU and softplus do not have this as their range are from 0 to infinity (non-negative), so they won't be centred around 0. However, most people still use ReLU as a reasonable default, while LReLU and ELU offer no benefit. It is a case-by-case basis, so have to experiment yourself.

There are also experimental activation functions such as the **bionodal root unit (BRU)** which has not caught on in the data science community. The author says that the BRU AF led to better results than ReLU and ELU.

Binary vs Multiclass Classification

In practice, we use ReLU for the hidden layers, and sigmoid for the output for binary classification ("Yes" vs "No").

For multiclass classifications, we do it a different way for the output.



Suppose we calculate the value just before applying the final activation function, such that we get $\mathbf{a_L = w_L^T * z_{L-1} + b_L}$ (read as value = transposed weight times the output from previous layer, plus bias). It is important to note that if the layer has size K, then a_L is a vector of size K, i.e. if the penultimate layer has 5 nodes in its layer, then the layer has a size of 5 and a_L is a vector with size of 5.

We must then map a_L to values of probabilities. They must be between 0 and 1, and yet must all sum to 1. There is the **softmax** function that does this. It basically exponentiates all values, then divide by the sum of all exponential values. This creates a probability distribution amongst the values in a. Note that the softmax function should only be applied on the final layer and not recommended on the hidden layers.

Conclusion on Activation Functions

For linear regression, no need any AF, just need to return the value calculated. For binary classification, use sigmoid. For multiclass classification, use softmax. Note that you can also use softmax for binary classification as well, but redundant.

Chapter 3.2: Classifying Images

RGB Format and Hex Colour Codes

Images are popularly stored in the **RGB format**. By mixing RGB in different proportions, we get different colours. Each value is stored as A(i,j,k) where i^{th} row, j^{th} column, and k^{th} colour (r/g/b). Images are now stored in 8 bits (= 1 byte), i.e. 00110011. Hence there are 2^8 = 256 possible values from 0 to 255, and in total with a 3D tensor we can store $2^8 \times 2^8 \times 2^8$ = 16.8 million colours.

For example, a 500x500 image will take up 500 x 500 x 3 x 8 = 6 million bits = 750,000 bytes = 732KB = about 1 MB. This is quite large, which is why thank god we have image compression algorithms like jpeg.

On a side note, the **hex colour codes** #FF00CC should be read as FF-00-CC, where each section is a hex number 00, 01, 02 etc. The digits used can be 0 to 9 or A to F. These are 16 values, so a hex number will give us 16 x 16 = 256 different possible values to represent one byte. Then we have 3 to match the RGB system.

As an extension, **grayscale images** are also 0 to 255 for each value, but is a 2D array only.

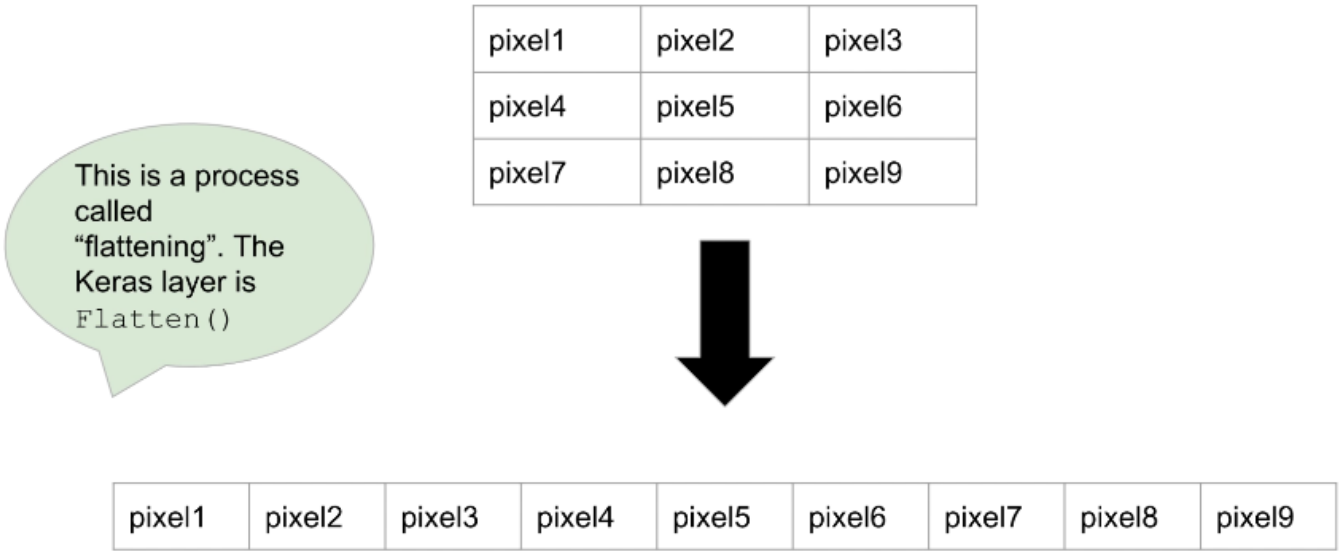
For coding-wise, you can use `plt.imshow(array2d)` to get a heatmap, where the blue shows minimum values and red shows max values. You can use `plt.imshow(array2d, cmap = 'gray')` for grayscale images.

Processing the Images in Neural Networks

For neural networks, since they do not like large values (0 to 255), we usually scale them to 0 to 1. Despite not centred around zero, this is an exception to the rule! It is also a convenient representation as they can, if needed, be interpreted as probabilities.

Another exception is the neural network for computer vision called **VGG**. It does not scale the input data, but does subtract the mean across each colour channel, so the images are centered around 0 but the range is 256. In TF, you can preprocess images using `tf.keras.applications.vgg16.preprocess_input`.

Flattening



Using Flatten(), we can convert a dataset of images into an NxD array, where N is the no. of samples while D is H x W x C in the RGB format.

Libraries

```
%tensorflow_version 2.x

import tensorflow as tf
print(tf.__version__)
```

Data

```
# We will be using MNIST dataset. Each image is 28
x 28 = 784 pixels grayscale.
# Each pixel is between 0 to 255. We will scale to
0 to 1.
# The data has already been segregated into train
and test sets for us.
# x_train.shape = N x 28 x 28; y_train.shape = N
# x_test.shape = Ntest x 28 x 28; y_test = Ntest

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) =
mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
print("x_train.shape:", x_train.shape)
```

Instantiate Model

```
# Flatten() to change from Nx28x28 to Nx784 so it
is a 2D matrix. WHY? Because it is more efficient.
# Dense()'s number of 128 is arbitrary.
# Dropout() is a layer that helps to regularise
the NN. It prevents the NN from depending on any
single input too strongly.
# It randomly drops input nodes by sending them to
zero so that they have no influence on the next
layer.
# Because it is random, the dropped nodes change
every time we use the data. This will teach the NN
to learn more evenly from all its inputs.
# In our code, there is a 20% of dropping a node
in that layer.
# The last Dense() number is the number of outputs
/ classes. Since there are 10 digits, it is 10.

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28,28)),
  tf.keras.layers.Dense(128, activation = 'relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

Compile the Model

```
# Loss function is sparse categorical
crossentropy.
# Cat XEntropy: If perfect prediction, then the
loss function returns 0. If wrong, then values can
go to infinity. For parts where target is 0, then
result will be 0.
# Sparse: Because of one-hot encoding, there are
many 0s except for that single 1.

model.compile(optimizer='adam',

loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train the Model

```
r = model.fit(x_train, y_train, validation_data =
(x_test, y_test), epochs = 10)
```

Plot Loss per Iteration

```
import matplotlib.pyplot as plt
plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
```

Plot Accuracy per Iteration

```
plt.plot(r.history['accuracy'], label='acc')
plt.plot(r.history['val_accuracy'],
label='val_acc')
plt.legend()
```

Evaluate the Model

```
print(model.evaluate(x_test, y_test))
```

Confusion Matrix

```
from sklearn.metrics import confusion_matrix
import numpy as np
import itertools

def plot_confusion_matrix(cm, classes,
                          normalize=False, # can be True
if you want to normalise
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
  if normalize:
    cm = cm.astype('float') / cm.sum(axis=1)[:,
np.newaxis]
    print("Normalised confusion matrix")
  else:
    print("Confusion matrix, without normalisation")
  print(cm)

  plt.imshow(cm, interpolation='nearest', cmap=cmap)
  plt.title(title)
  plt.colorbar()
```

Show some Misclassified Examples

```
# Misclassified Indices

misclassified_idx = np.where(p_test != y_test)[0]
i = np.random.choice(misclassified_idx)
plt.imshow(x_test[i], cmap='gray')
plt.title("True label: %s Predicted: %s" % (y_test[i],
p_test[i]));
```

```
  tick_marks = np.arange(len(classes))
  plt.xticks(tick_marks, classes, rotation=45)
  plt.yticks(tick_marks, classes)

  fmt = '.2f' if normalize else 'd'
  thresh = cm.max() / 2.
  for i, j in itertools.product(range(cm.shape[0]),
range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], fmt),
             horizontalalignment='center',
             color='white' if cm[i,j] > thresh else
"black")

  plt.tight_layout()
  plt.ylabel('True label')
  plt.xlabel('Predicted label')
  plt.show()

p_test = model.predict(x_test).argmax(axis=1)
cm = confusion_matrix(y_test, p_test)
plot_confusion_matrix(cm, list(range(10)))
```

Chapter 4: Convolution Neural Networks (CNN)

Chapter 4.1: Introduction to Convolution

CNNs are used in pretty advanced stuff like signal processing and computer vision. However, it is mainly just two things: addition and multiplication.

An easy way to understand is if you **convolve (*)** an input image with a **filter / kernel**, you will get an output image.

Similarly, an example of convolution is to blur images, or edge detection. In other words, convolution is like **feature transformation** on the image – which sounds very alike to neural networks, doesn't it! What makes one convolution (blur) different from another (edge detection) is the **filter / kernel**.

Mechanics of Convolution with Simple Math

Image * Filter

0	10	10	0
20	30	30	20
10	20	20	10
0	5	5	0

1	0
0	2

Image * Filter = Output

0	10	10	0
20	30	30	20
10	20	20	10
0	5	5	0

1	0
0	2

60		

$$1*0 + 0*10 + 0*20 + 30*2 = 60$$

Image * Filter = Output

0	10	10	0
20	30	30	20
10	20	20	10
0	5	5	0

1	0
0	2

60	70	

$$1*10 + 0*10 + 0*30 + 2*30 = 70$$

Image				*	Filter		=	Output		
0	10	10	0		1	0		60	70	50
20	30	30	20		0	2		60	70	50
10	20	20	10					20	30	20
0	5	5	0							

Hence, if the input length = N, and kernel length = K, then output length = N – K + 1. Take note that images may not be always square, while filters / kernels are always square by convention.

The convolution equation is as below, where it shows the (i,j)th entry of the output. For the second equation, Z is the output, X is the filter, and Y is the input.

$$(A * w)_{ij} = \sum_{i'=1}^{K} \sum_{j'=1}^{K} A(i+i', j+j') w(i', j')$$

$$Z(i,j) = \sum_{m=1}^{M} \sum_{n=1}^{N} X(m,n) Y(i-m, j-n)$$

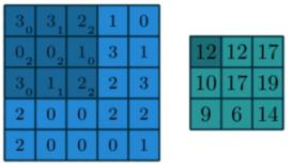
On another hand, these are other equations of convolution. Of all four, which is correct? They are all correct.

True convolution $$(A * w)_{ij} = \sum_{i'=1}^{K} \sum_{j'=1}^{K} A(i-i', j-j') w(i', j')$$

Deep learning convolution $$(A * w)_{ij} = \sum_{i'=1}^{K} \sum_{j'=1}^{K} A(i+i', j+j') w(i', j')$$

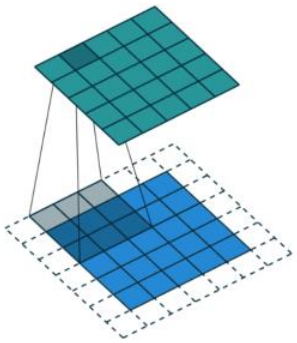
In a sense, what we are *actually* doing in deep learning is called **cross-correlation**, where if I say "X and Y are correlated" means there is a degree of similarity between X and Y. Hence, CNN can be thought of as a "correlation neural network", just that there is flipping around in convolution; there is reversal of the orientation of filter.

Mode

Normal	Padding ("same" mode)
Normal / "Valid"	• `convolve2d(A, np.fliplr(np.flipud(w)), mode='valid')` • The movement of the filter is bounded by the edges of the image ○ The output is therefore always smaller than the input 

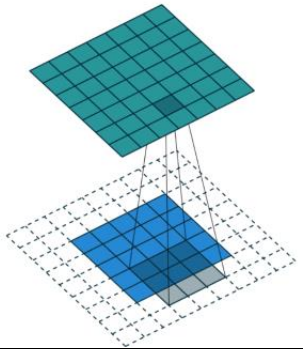
Padding ("same" mode)

- What if we want the output to be the **same size** as the input?
- Then we can use padding (add imaginary zeros around the input)



Padding ("full" mode)

- We *could* extend the filter further and still get non-zero outputs
- Are we losing information if we don't?
- "Full" padding:
  - Input length = N
  - Kernel length = K
  - Output length = N + K - 1



Summary

- Input length = N, Kernel length = K

Mode	Output Size	Usage
Valid	N - K + 1	Typical
Same	N	Typical
Full	N + K - 1	Atypical

<u>The Math Behind It</u>

Remember that we said that convolution is just like correlation? Convolution uses the **dot product**. This dot product can also be seen as **cosine similarity / cosine distance**, which in turn is very similar to the **Pearson correlation**.

- We like to vectorize operations, because Numpy functions are faster than Python for loops
- We can always simplify:

$$a^T b = \sum_{i=1}^{N} a_i b_i$$

They would be exactly the same if we flattened the convolution inputs

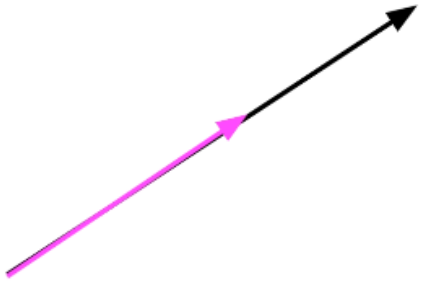
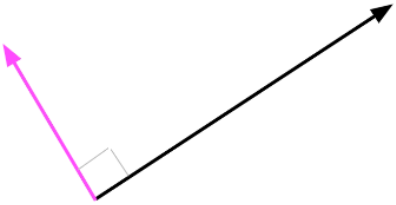
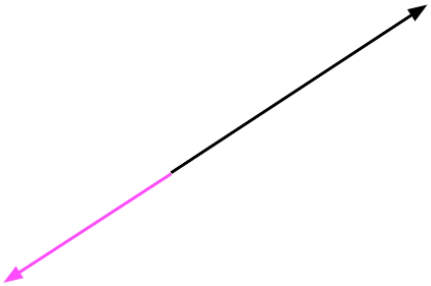
- Convolution kind of looks like this:

$$(A * w)_{ij} = \sum_{i'=1}^{K} \sum_{j'=1}^{K} A(i+i', j+j') w(i', j')$$

Dot product:

$$a \bullet b = \sum_{i=1}^{N} a_i b_i = |a||b| \cos \theta_{ab}$$

Appreciating cosine. So cos(a,b) is just asking: how similar are a and b? The larger the value, the closer they are; the smaller the value, the further they are.

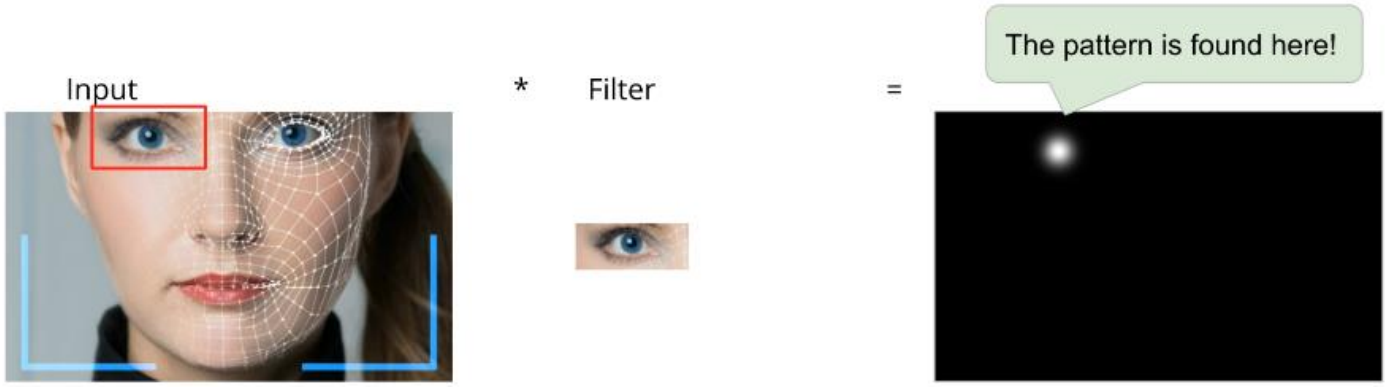
cos(0) = 1 (max value of cosine)	cos(90) = 0 (orthogonal)	cos(180) = -1 (min value of cosine)
		

And here is the cosine equation rearranged, and the **Pearson correlation** which is same but with mean subtraction.

$$\cos(a,b) = \frac{\sum_{i=1}^{N} a_i b_i}{\sqrt{\sum_{i=1}^{N} a_i^2}\sqrt{\sum_{i=1}^{N} b_i^2}} = \frac{a \bullet b}{|a||b|} \varrho_{ab} = \frac{\sum_{i=1}^{N} (a_i - \bar{a})(b_i - \bar{b})}{\sqrt{\sum_{i=1}^{N} (a_i - \bar{a})^2}\sqrt{\sum_{i=1}^{N} (b_i - \bar{b})^2}}$$

So ultimately, you can think of the dot product as a correlation measure: if high positive correlation, dot product is large and positive; if high negative correlation, dot product is large and negative; if no correlation (orthogonal), then the dot product is zero.

The reason why this is important is because you don't have to think of filters as an abstract concept – it is merely just a that filters out everything not related to the pattern contained in the filter. It slides across the whole image and gives a high value if pattern is similar, and a low value if pattern is not.



<u>Another Perspective of Convolution</u>

Using a 1D convolution to show the proof of concept that convolution is *<u>like</u>* matrix multiplication:

1-D convolution

- Easier to see if we just do a 1-D convolution
- Input image: a = [a_1, a_2, a_3, a_4]
- Filter: w = [w_1, w_2]
- Output image: b = a * w = [$a_1w_1 + a_2w_2$, $a_2w_1 + a_3w_2$, $a_3w_1 + a_4w_2$]

- We can summarize it as follows
- Same as 2-D convolution, just without 2nd index

$$b_i = \sum_{i'=1}^{K} a_{i+i'} w_{i'}$$

- We can do the same operation using matrix multiplication

$$\begin{pmatrix} a_1w_1 + a_2w_2 \\ a_2w_1 + a_3w_2 \\ a_3w_1 + a_4w_2 \end{pmatrix} = \begin{pmatrix} w_1 & w_2 & 0 & 0 \\ 0 & w_1 & w_2 & 0 \\ 0 & 0 & w_1 & w_2 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix}$$

b w a

- Why would we *not* prefer matrix multiplication?
- It takes up too much space!
- Filter w was a vector of size 2
- This matrix is 3x4!

$$\begin{pmatrix} a_1w_1 + a_2w_2 \\ a_2w_1 + a_3w_2 \\ a_3w_1 + a_4w_2 \end{pmatrix} = \begin{pmatrix} w_1 & w_2 & 0 & 0 \\ 0 & w_1 & w_2 & 0 \\ 0 & 0 & w_1 & w_2 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix}$$

Parameter Sharing / Weight Sharing

- What if, instead of a full weight matrix, we only used the same 2 weights over and over?
- Then, we could have less parameters, use up less RAM, and make the computation more efficient

Why do this?

- Consider a fully-connected ANN
- MNIST: 28 x 28 = 784-sized input vector
- CIFAR-10: 32 x 32 x 3 = 3072-sized input vector
- 32 x 32 is a very modest size
- VGG (a modern CNN) looks at images of size 224 x 224
  - 150,528 features
- HD Image: 1280 x 720
  - 2.8 million features

$$a = W^T x$$

Why do this?

- Remember, convolution is a *pattern finder*
- We want the **same** filter to look at **all locations** in the image
- This is the idea behind "translational invariance"

Translational Invariance

- Suppose we're building a dog vs. cat recognizer

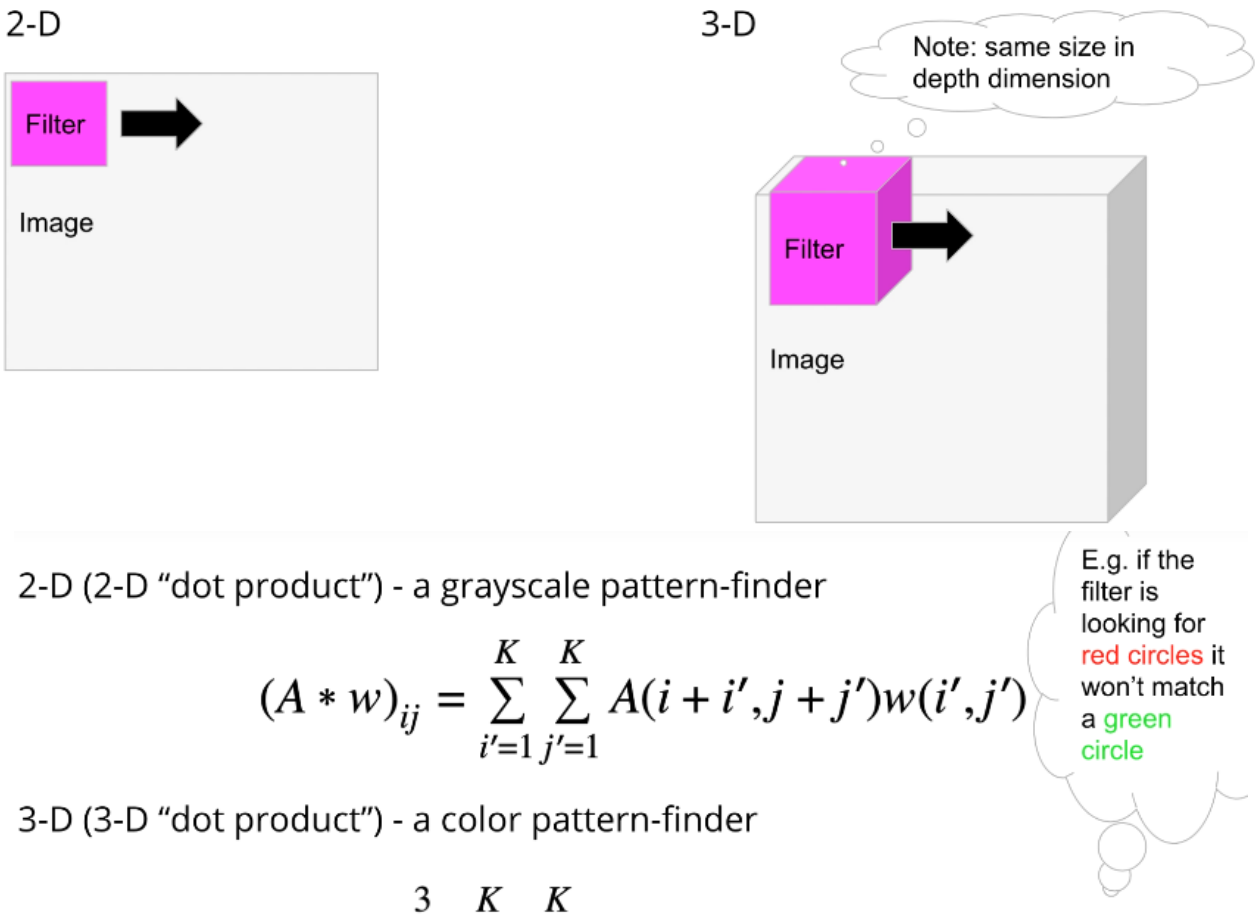
If we used a fully-connected (dense) neural network, the weights would have to learn to find the cat in each possible position separately!

Cat Cat

Therefore, we use CNNs on images because it takes up less resources. Furthermore, even if the same cat is in a different position, a normal dense NN will fail to recognise it. Hence, it is better to have a **shared pattern finder** because it looks at all locations on the image – it does not need to learn the weights to look at a cat at every single possible point which is very infeasible.

Chapter 4.2: Convolution on Colour Images

In the previous examples, we use a 2D convolution. But for colour images, they are usually 3D, so we now use a 3D filter for this 3D image.



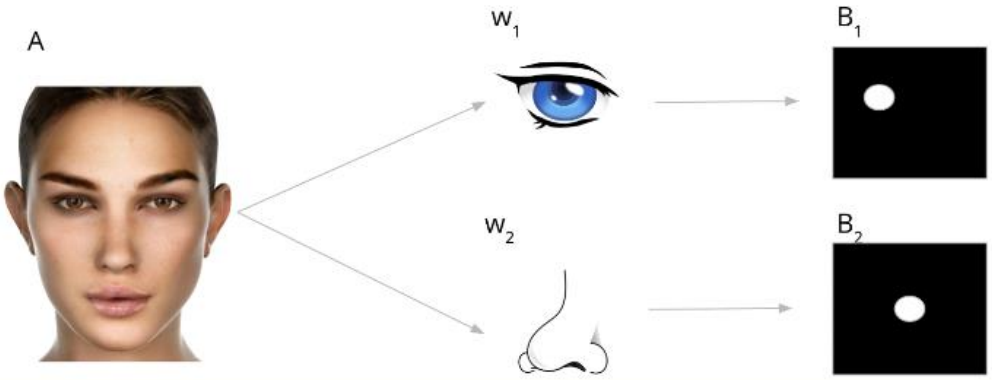
The below slide helps to explain that reading a 3D image with a 3D filter, it will produce a 2D output image. However, this will affect the uniformity of the neural network, e.g. for a Dense layer, if input is 1D the output should be 1D as well, and hence this 1D output can be fed to the next layer as 1D, and so on… This does not happen for convolution so the process may be disrupted.

- Input image: H x W x 3
- Kernel: K x K x 3
- Output image: (H - K + 1) x (W - K + 1)
- This should offend your sensibilities
- Neural networks have repeating structures ("uniformity")
  - E.g. for Dense layer, input is a 1-D vector, output is also a 1-D vector
  - Therefore, can be fed into another Dense layer
- But if output is 2-D, how can we do another convolution later?

$$(A * w)_{ij} = \sum_{c=1}^{3} \sum_{i'=1}^{K} \sum_{j'=1}^{K} A(i+i', j+j', c)w(i', j', c)$$

However remember that for CNNs on images, we are looking for multiple features, e.g. we are looking for an eye, a nose etc. Hence we have different filters working on the same image. And as below, we can make this work:

- A.shape = H x W x 3
- If we use "same mode", then B_1.shape = H x W, B_2.shape = H x W
- If we stack B_1 and B_2, we get B.shape = H x W x 2
- We can add any number of features!



Now, the input will be the same as output. In fact, the output can grow to any size – it depends on how many filters we use! We want to vectorise the operation so we can do all in one go rather than doing many different convolutions and then stacking them together – both input and outputs are 3D now.

- Let's vectorize this operation; we don't need to do each "color convolution" separately

$$B = A * w$$

$$shape(A) = H \times W \times C_1$$

$$shape(w) = C_1 \times K \times K \times C_2$$

$$shape(B) = H \times W \times C_2$$

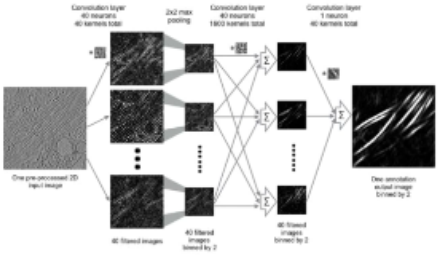
$$B(i,j,c) = \sum_{i'=1}^{K} \sum_{j'=1}^{K} \sum_{c'=1}^{C_1} A(i+i', j+j', c') w(c', i', j', c)$$

Summary So Far

- We realized we previously only defined convolution for 2-D (grayscale) images and corresponding 2-D filters
- We extended this to color images by saying the filter should just have the same depth, and then we "dot" along all 3 axes
- This breaks uniformity, because input is 3-D but output is still 2-D
- So we wouldn't be able to stack multiple convolutions sequentially
- We extended this further by noting that each layer should find multiple features (using multiple filters)
- This results in multiple 2-D outputs (one per filter) which we can stack to get out a 3-D image once again

While a normal coloured image is HxWx3, the 'colour' column does not make sense after convolutions as it is instead the number of "stacks". This output is a stack of several **feature maps**, one map for each feature.

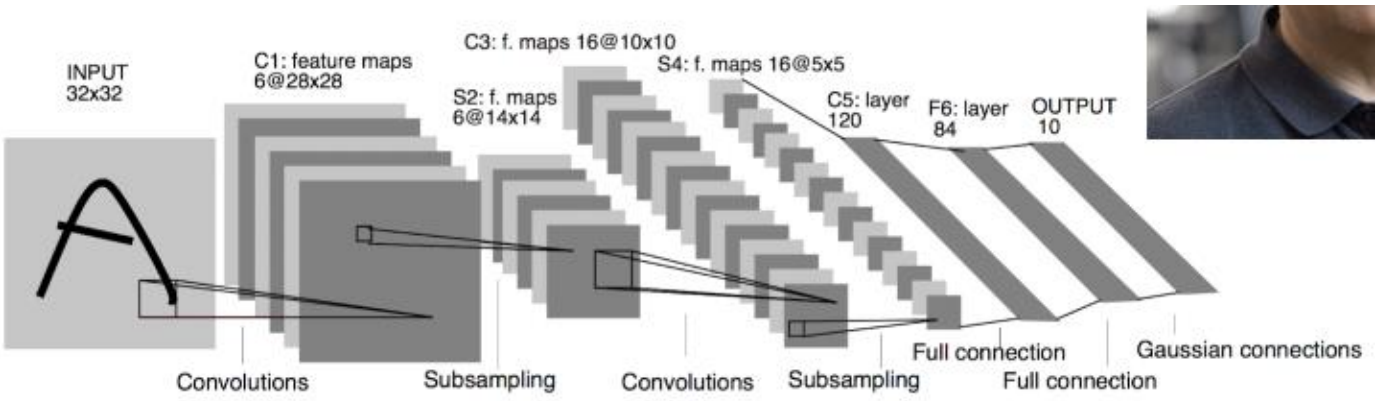
- Input to the neural network is a true color image: H x W x 3
- But after subsequent convolutions, we'll just have H x W x (arbitrary #)
- Our terminology must be more general since the 3rd dimension no longer necessarily represents color
- We call these "feature maps" (e.g. each 2-D image is a *map* that tells us *where the feature is found*)
- We thus call the size of the final dimension the:
  - Number of channels
  - Number of feature maps



<u>The Convolution Layer</u>

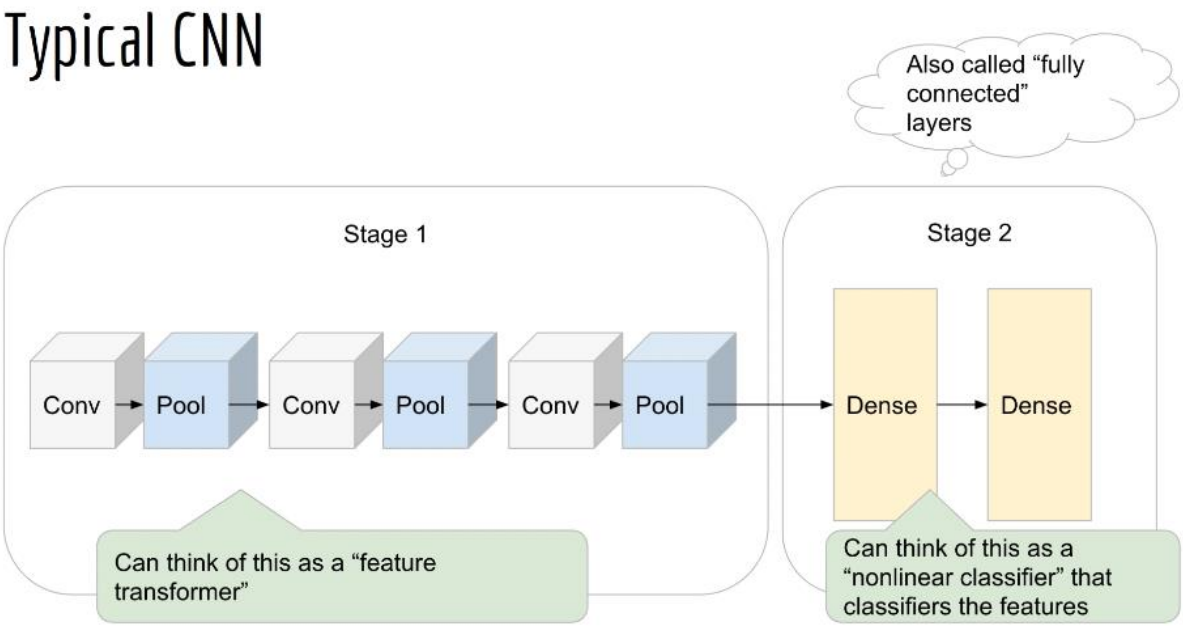
- Similar to the Dense() layer, it is a shared-weight version of matrix multiplication. Not similar to Dense() is that it is **parameter sharing**, meaning it uses the same weights in multiple places, reducing the number of total weights we need.
- However, while Dense()'s bias term has same shape as w^Tx to allow matrix addition, Conv()'s bias term **<u>does not</u>** have the same shape as its w*x. Its w*x is a 3D image and if it has a shape of $HxWxC_2$, then b will be a vector of size C_2. This is usually not allowed in matrix addition (since different shape), but the rules of broadcasting in Numpy allows it so. Hence, bias is a vector of size C_2 (the no. of feature maps), with one scalar per feature map.
- Example:
  - Convolution Layer (parameter sharing)
    - Input Image: 32x32x3
    - Filter: 3x5x5x64 (so it is looking for 64 features)
    - Output Image: 28x28x64 (32 – 5 + 1 = 28, using valid mode convolution)
    - # of parameters (ignoring bias term) = 3x5x5x64 = 4800
  - Feedforward Dense Layer (full matrix multiplication)
    - Flattened Input Image: 32x32x3 = 3072
    - Flattened Output Vector: 28x28x64 = 50176
    - Weight Matrix: 3072x50176 = 154,140,672 = 154 million parameters
  - Hence, convolution's parameter sharing helps to save a lot of resources! It works for convolution because we <u>want</u> to use the **<u>same pattern finder in multiple places</u>**.
- The filter still applies the usual gradient descent method when we call model.fit().

Chapter 4.3: CNN Architecture



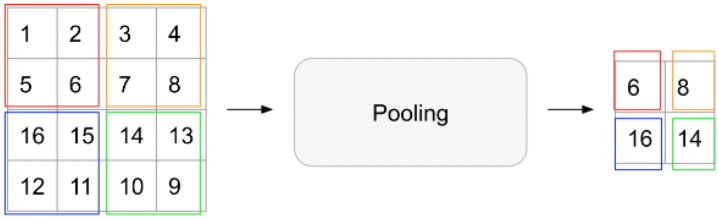
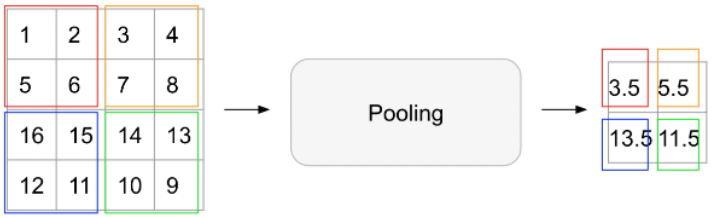
A typical CNN has two stages:

- Stage 1: series of convolution layers and **pooling layers**. It is like a feature transformer that specifically works on images and find image features.
- Stage 2: series of dense layers called **fully connected layers** (regular feedforward NN). The output of stage 1 are then subjected to nonlinear classification or regression here.

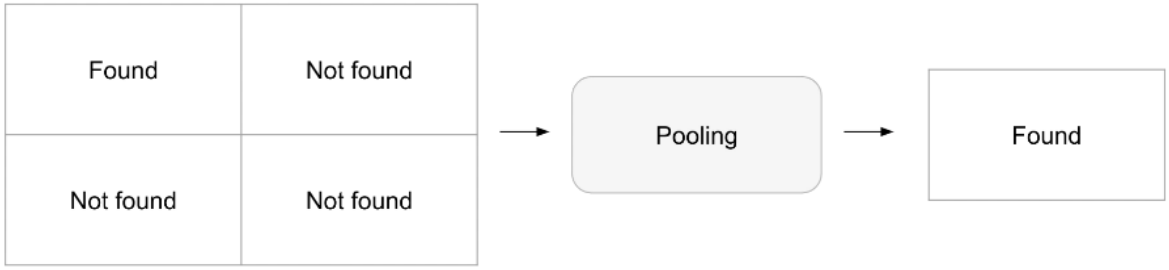


Pooling

- At a high level, pooling is **downsampling**, e.g. make a smaller output image from a bigger image.
- If an input is 100x100, a pool size of 2 would yield 50x50.
- The purposes are: ① to shrink the image so have less data to process, ② **translation invariance** where I don't care where in the image the feature occurred, I just care that it did.
- There are two types of pooling: **max** and **average**:

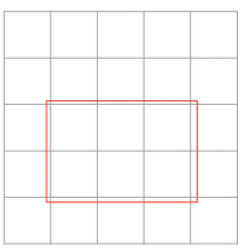
Max Pooling	Average Pooling
	
• Chooses the maximum value from each square • More commonly used	• Takes the average of each square

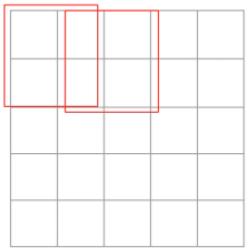
Recall that convolution is a pattern finder, where the highest number is the best matching location. Hence, using either max or average pooling, if a feature is found in an image, it will return a high number and be "found" while the rest are low numbers and are "not found". By pooling, it tells me that the pattern is found without caring where it was found (translation invariance):



It is more common for the pool sizes to be a square of 2x2, and for them to have a **stride** of 2, so that there are no overlaps. However, there are alternatives available (not really used) to have a non-square window or overlapping boxes:

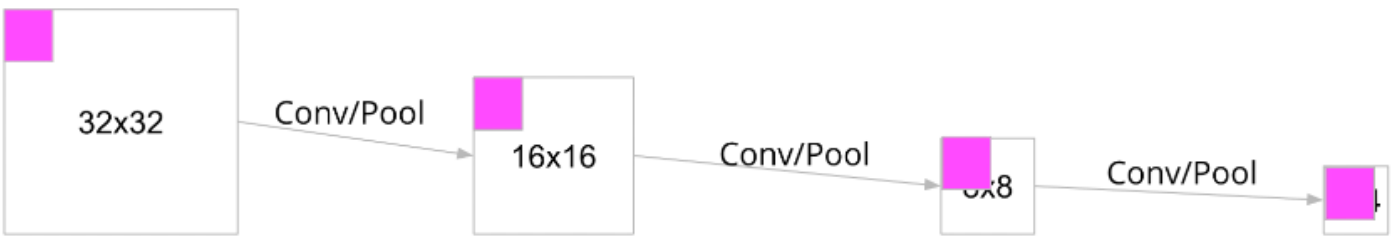
- It's possible to have a non-square window, e.g. 2x3 or 3x2, but this is unconventional
- It's also possible for boxes to overlap (this is called "stride")
  - Previously, we looked at a pool size of 2 with a stride of 2 (common)
  - If you had a stride of 1, the boxes would overlap (not common)





Q: So why convolution + pooling? A: First, recall that CNN learns features hierarchically so the initial layers end up learning basic strokes, the next layer ends up learning individual facial features (nose eyes lips), and the next layer ends up learning whole faces, as seen in the below picture:

The key point is that after each conv-pool, the image shrinks but the filter sizes generally stay the same (commonly 3x3, 5x5, or 7x7). So assume in our below example that we have 4 conv-pool layers for an input image of 32x32, using "same mode" convolution with filters size 3x3 and pool size = 2. Since the filter size stays the same but the image shrinks, then the portion of the image that the filter covers increases – the filters find increasingly large patterns relative to the image, and this is why it is **learning hierarchical features of the input**! Hence, at 32x32, it was still finding general strokes, but at 16x16 it was detecting some facial features etc.



One disadvantage is that we are losing information since we are shrinking the image – losing spatial information since we do not care where the feature was found. However, this is because we have not yet considered the number of feature maps – while the image size decreases, the number of feature maps increases – you are gaining info.

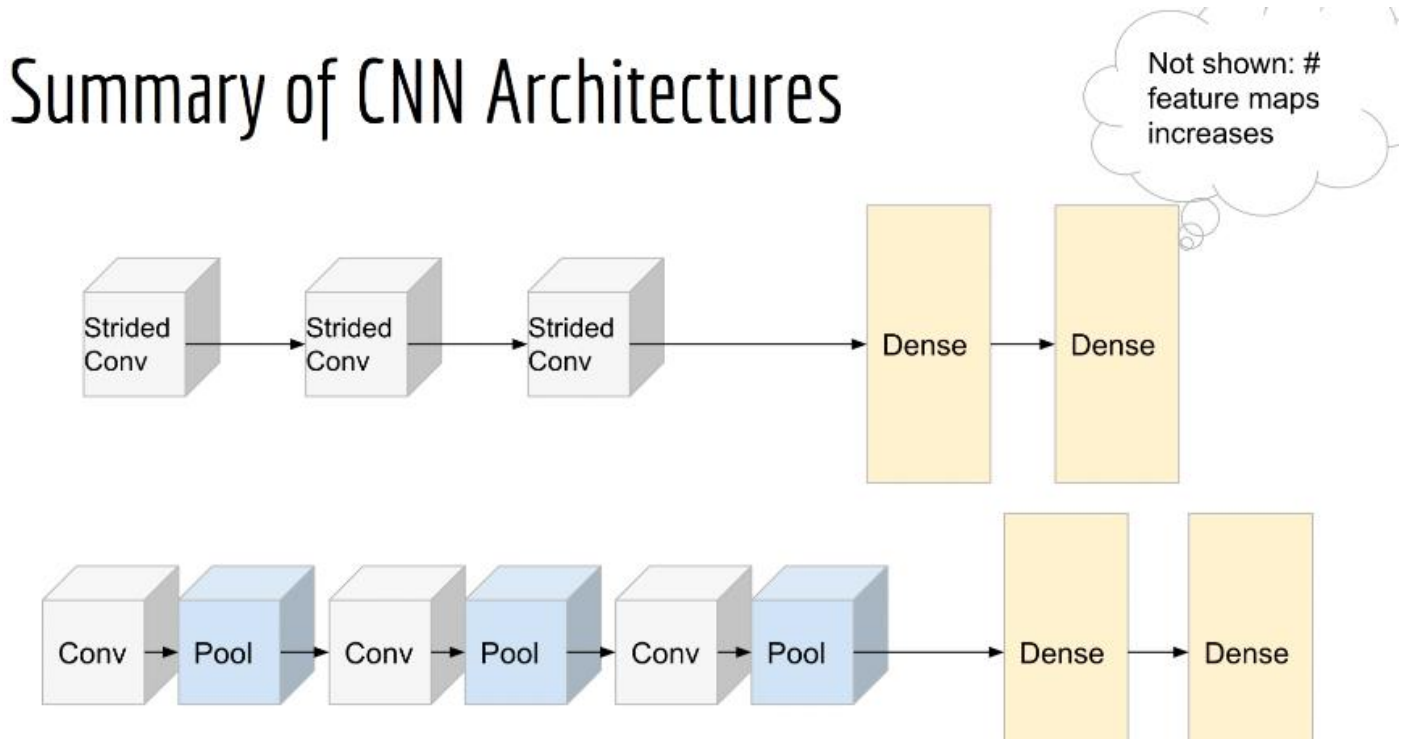
Hyperparameters

- Previously in ANNs: learning rate, # hidden layers, # hidden nodes per layer
- With CNNs: conventions are pretty standard
    - Small filters relative to image: 3x3, 5x5, 7x7
    - Repeating of conv > pool > conv > pool etc.
    - Increase # feature maps, e.g. 32 > 64 > 128 > 128 again etc.

Alternative to Pooling – Stride

- Researchers have found that sometimes we can avoid pooling and just do strided convolution instead.
- Recall that stride tells us how far apart each box should be when we take the max.
- Convolution has a stride option as well! Remember that convolution just means multiply and add with a sliding window. When we have convolution with a stride parameter, the stride tells us how far apart each window should be. Hence if stride is two, then the output image length is half of original. In other words, we get the same output with stride or with pooling.
- For a normal picture, the pixels around it should be similar to its colour, i.e. a red pixel's neighbours are probably also red. A stride of 2 simply means having your filter skipping every other pixel, because we don't care what those skipped pixels are because they probably are very close to the pixels we were already looking at.

See below:



Stage 2: Dense Layers

- The output of a convolution is a 3D HxWxC (where C = # feature maps) object, but a feedforward dense layer expects a 1D input vector.
- Hence, we Flatten() the 3D object in Keras first before feeding it in.
- One alternative to Flatten() is the **global max pooling layer**. This is **optional**. Imagine if your input images are all of different size – how does your CNN handle this? It uses the global max pooling layer.
    - Flatten() not good because of the below.
    - Imagine if you do 4 convolutions with stride = 2, then a 32x32 input image will become 32x32 > 16x16 > 8x8 > 4x4 > 2x2; while your 64x64 > 32x32 > 16x16 > 8x8 > 4x4. Then if you Flatten() and assuming 100

feature maps, you will get a 2x2x100 = 400-D vector and a 4x4x100 = 1600-D vector respectively. A feedforward neural network cannot handle input vectors of different sizes!

- Global Max Pooling **always** outputs a 1x1xC matrix (which TF will change to a vector of only size C), regardless of H & W. It takes the **max** over each feature map. Since it has C feature maps, then you will end up with a one-dimensional vector of size C since the 1 dimensions are redundant. It follows the same principle of "don't care where it is found, as long as it is found".
- Also exists a **global average pooling**.

- An exception is images that are too small, e.g. a 2x2 image cannot be halved 4 times. This happens when there are too many conv layers in your CNN.

Summary

Step #1:

- Conv > Pool > Conv > Pool > ...
- Strided Conv > Strided Conv > ...

Step #2:

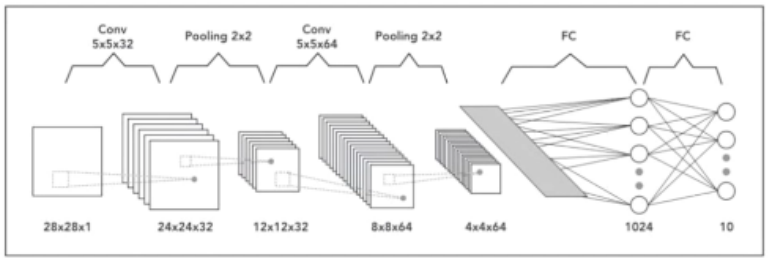
- Flatten()
- GlobalMaxPooling2D()

Step #3:

- Dense > Dense > ...

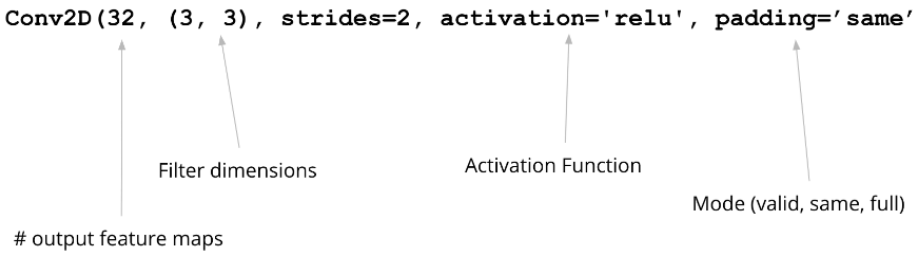
As usual, the 3 tasks determine the final activation / # of nodes:

- Regression: none
- Binary Classification: sigmoid
- Multiclass Classification: softmax



- Note: it's Conv2D because there are 2 spatial dimensions
- Also have Conv1D and Conv3D
- A time-varying signal would use Conv1D
- A video (Height, Width, Time) would use Conv3D
- Voxels (Height, Width, Depth) would use Conv3D
- E.g. medical imaging data
- Pixel = "Picture Element"
- Voxel = "Volume Element"

Chapter 4.4: CNN Code



Fashion MNIST

Import Libraries

```
%tensorflow_version 2.x

import tensorflow as tf
print(tf.__version__)

# We do the below so later don't need to keep on
typing tf.keras.layers....

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Input, Conv2D,
Dense, Flatten, Dropout
from tensorflow.keras.models import Model
```

Load Data

```
# fashion_mnist is a Nx28x28 grayscale images, so
pixel values are 0 to 255
# Need to reshape to NxHxWxC (4D) for CNN

fashion_mnist = tf.keras.datasets.fashion_mnist

(x_train, y_train), (x_test, y_test) =
fashion_mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
print("x_train.shape:", x_train.shape)

# Reshaping Data (CNN expects HxWxC). We use -1 to
refer to new axis.

x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
print(x_train.shape)

# Number of Classes
# set() returns unique values from an iterable

K = len(set(y_train))
print("number of clases:", K)
```

Modelling

```
# Note the increasing # of feature maps per layer, a
constant size filter, and stride of 2 to half pics.

i = Input(shape=x_train[0].shape)
x = Conv2D(32, (3, 3), strides = 2,
activation='relu')(i)
x = Conv2D(64, (3, 3), strides = 2,
activation='relu')(x)
x = Conv2D(128, (3, 3), strides = 2,
activation='relu')(x)
x = Flatten()(x)
x = Dropout(0.2)(x)
x = Dense(512, activation='relu')(x)
x = Dropout(0.2)(x)
x = Dense(K, activation='softmax')(x)

model = Model(i, x) # Model = (input, output)
```

Compile & Fit

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
r = model.fit(x_train, y_train,
validation_data=(x_test, y_test), epochs = 15)

# Plot Loss per Iteration

# Notice how the plot the validation loss is going up,
means we have been overfitting.

plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()

# Plot Accuracy per Iteration

plt.plot(r.history['accuracy'], label='acc')
plt.plot(r.history['val_accuracy'], label='val_acc')
plt.legend()

# From both loss and accuracy graphs, we can see that
the accuracy remains steady when loss goes up.
# This could mean that the model is becoming more and
more confident in its incorrect predictions.
# When predictions become more incorrect, the loss gets
bigger. But a prediction becoming incorrect when it was
already incorrect would not affect accuracy.
# E.g. a cat was predicted to be 0.6 horse and 0.4 cat
(incorrect). The next round, it was predicted 0.7 horse
and 0.3 cat (incorrect). The loss gets bigger, but the
accuracy unaffected.
```

Label Mapping & Misclassified Samples

```
labels = '''T-shirt/top
Trouser
Pullover
Dress
Coat
Sandal
Shirt
Sneaker
Bag
Ankle boot'''.split()

misclassified_idx = np.where(p_test != y_test)[0]
i = np.random.choice(misclassified_idx)
plt.imshow(x_test[i].reshape(28,28), cmap='gray')
plt.title("True label: %s Predicted: %s" %
(labels[y_test[i]], labels[p_test[i]]));
```

CIFAR-10

Libraries

```
%tensorflow_version 2.x
import tensorflow as tf
print(tf.__version__)

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Input, Conv2D,
Dense, Flatten, Dropout, GlobalMaxPooling2D
from tensorflow.keras.models import Model
```

Data

```
# CIFAR10 has 10 classes and is of Nx32x32x3.
# Its labels / targets / y are Nx1 so need to
flatten() it.

cifar10 = tf.keras.datasets.cifar10

(x_train, y_train), (x_test, y_test) =
cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train, y_test = y_train.flatten(), y_test.flatten()
print("x_train.shape:", x_train.shape)
print("y_train.shape:", y_train.shape)

# Number of Classes
K = len(set(y_train))
print("Number of classes:", K)
```

Modelling

```
# Not seen here is that the first filter is size
1x3x3x32 if only 1 color channel, and 3x3x3x32 if 3.
# Remember that filter size is C1color x K x K x
C2features

i = Input(shape=x_train[0].shape)
x = Conv2D(32, (3, 3), strides=2,
activation='relu')(i)
x = Conv2D(64, (3, 3), strides=2,
activation='relu')(x)
x = Conv2D(128, (3 ,3), strides=2,
activation='relu')(x)
x = Flatten()(x)
x = Dropout(0.5)(x)
x = Dense(1024, activation='relu')(x)
x = Dropout(0.2)(x)
x = Dense(K, activation='softmax')(x)

model = Model(i, x)
```

Compile & Fit

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
r = model.fit(x_train, y_train,
validation_data=(x_test, y_test), epochs=15)

# Plot Loss per Iteration

plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
```

```
# Plot Accuracy per Iteration

plt.plot(r.history['accuracy'], label='acc')
plt.plot(r.history['val_accuracy'], label='val_acc')
plt.legend
```

Confusion Matrix

```
from sklearn.metrics import confusion_matrix
import numpy as np
import itertools

def plot_confusion_matrix(cm, classes,
                          normalize=False, # can be
True if you want to normalise
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
  if normalize:
    cm = cm.astype('float') / cm.sum(axis=1)[:,
np.newaxis]
    print("Normalised confusion matrix")
  else:
    print("Confusion matrix, without normalisation")
  print(cm)

  plt.imshow(cm, interpolation='nearest', cmap=cmap)
  plt.title(title)
  plt.colorbar()

  tick_marks = np.arange(len(classes))
  plt.xticks(tick_marks, classes, rotation=45)
  plt.yticks(tick_marks, classes)

  fmt = '.2f' if normalize else 'd'
  thresh = cm.max() / 2.
  for i, j in itertools.product(range(cm.shape[0]),
range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], fmt),
             horizontalalignment='center',
             color='white' if cm[i,j] > thresh else
"black")

  plt.tight_layout()
  plt.ylabel('True label')
  plt.xlabel('Predicted label')
  plt.show()

p_test = model.predict(x_test).argmax(axis=1)
cm = confusion_matrix(y_test, p_test)
plot_confusion_matrix(cm, list(range(10)))
```

Labels & Misclassified Examples

```
labels = '''airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck'''.split()

misclassified_idx = np.where(p_test != y_test)[0]
i = np.random.choice(misclassified_idx)
plt.imshow(x_test[i], cmap='gray')
plt.title("True label: %s Predicted: %s" %
(labels[y_test[i]], labels[p_test[i]]));
```

Chapter 4.5: Data Augmentation

- A cat can on be the left side of a picture, but a NN may not recognise it if it is on the right side of a picture. Hence, we can apply **data augmentation** to help the NN to recognise it.
- A disadvantage of older algorithms is that you **cannot invent new data** as it would be contaminated with your own biases, e.g. cannot invent a student that plays X hours of games and study Y hours to predict if pass or fail.
- But with images, **it is okay to invent new data** – you can put the cat anywhere, or you can flip it around. But the more data you invent, the more space it takes up! You can shift the picture by 1, 2, 3, 4, 5 pixels, rotate etc.
- But with the help of TF Keras's **generators / iterators**, we can do this without storing such images.

Generators / Iterators

In Python 3, we can use the **yield()** command so no list is ever created, and all values do not need to be stored in memory simultaneously since once it is used, it will iterate.

We can use this concept in Keras to generate augmented data on the fly. We will use the **ImageDataGenerator()** and the **flow()** method:

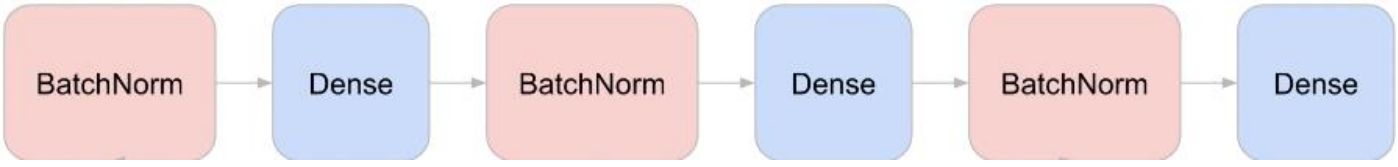
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

data_generator = ImageDataGenerator(
        width_shift_range = 0.1,
        height_shift_range = 0.1,
        horizontal_flip = True)
```

Other arguments: `rotation_range, width_shift_range, height_shift_range, brightness_range, shear_range, zoom_range, horizontal_flip, and vertical_flip.`

```
train_generator = data_generator.flow(
        x_train, y_train, batch_size)

steps_per_epoch = x_train.shape[0] // batch_size
r = model.fit_generator(
        train_generator,
        steps_per_epoch = steps_per_epoch,
        epochs=50)
```

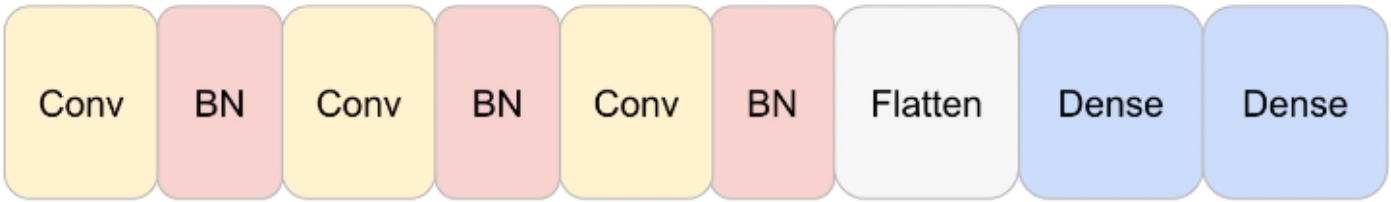
Chapter 4.6: Normalisation / Standardisation

In NN, a problem arises when you want normalised data, but after the first layer sigmoid, the output will not be normalised anymore. We can solve this problem using **batch normalisation**. This means that inserting such a layer would look at each batch, calculate its mean and SD, and standardise based on that.



One advantage is that it acts as **regularisation** to prevent overfitting. Since each batch is slightly different, you will get a slightly different mean and SD, which is essentially noise, and using noise during training makes the NN impervious to noise (aka not fitting to the noise / prevents overfitting).

Batch normalisation is not used in Dense(), but rather only in CNNs. You must choose whether to use it, and if so where in the process to layer it. They are usually placed after each Conv layer.



Improved CIFAR-10

Libraries

```
### Libraries

%tensorflow_version 2.x
import tensorflow as tf
print(tf.__version__)

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Input, Conv2D,
Dense, Flatten, Dropout, GlobalMaxPooling2D,
MaxPooling2D, BatchNormalization
from tensorflow.keras.models import Model
```

Data

```
cifar10 = tf.keras.datasets.cifar10

(x_train, y_train), (x_test, y_test) =
cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train, y_test = y_train.flatten(), y_test.flatten()
print("x_train.shape:", x_train.shape)
print("y_train.shape:", y_train.shape)

# Number of Classes
K = len(set(y_train))
print("Number of classes:", K)
```

Modelling

```
# Removed those Conv2D with strides. Instead, we will
use global max pooling. Also the structure is inspired
by the VGG network where we do multiple Convs before
pooling.
# Conv uses same padding. If not, image will shrink
too small with the number of convolutions.
# Also included Batch Normalisation.
# Most Dropout()s removed. May be weird to apply to
images, but can try.

i = Input(shape=x_train[0].shape)
x = Conv2D(32, (3, 3), activation='relu',
padding='same')(i)
x = BatchNormalization()(x)
x = Conv2D(32, (3, 3), activation='relu',
padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(64, (3, 3), activation='relu',
padding='same')(x)
x = BatchNormalization()(x)
x = Conv2D(64, (3, 3), activation='relu',
padding='same')(x)
x = BatchNormalization()(x)
```

```
# Plot Accuracy per Iteration

plt.plot(r.history['accuracy'], label='acc')
plt.plot(r.history['val_accuracy'], label='val_acc')
plt.legend

# Plot Accuracy per Iteration

plt.plot(r.history['accuracy'], label='acc')
plt.plot(r.history['val_accuracy'], label='val_acc')
plt.legend
```

Confusion Matrix

```
from sklearn.metrics import confusion_matrix
import numpy as np
import itertools

def plot_confusion_matrix(cm, classes,
                          normalize=False, # can be
True if you want to normalise
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
  if normalize:
    cm = cm.astype('float') / cm.sum(axis=1)[:,
np.newaxis]
    print("Normalised confusion matrix")
  else:
    print("Confusion matrix, without normalisation")
  print(cm)

  plt.imshow(cm, interpolation='nearest', cmap=cmap)
  plt.title(title)
  plt.colorbar()

  tick_marks = np.arange(len(classes))
  plt.xticks(tick_marks, classes, rotation=45)
  plt.yticks(tick_marks, classes)

  fmt = '.2f' if normalize else 'd'
  thresh = cm.max() / 2.
  for i, j in itertools.product(range(cm.shape[0]),
range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], fmt),
             horizontalalignment='center',
             color='white' if cm[i,j] > thresh else
"black")

  plt.tight_layout()
  plt.ylabel('True label')
  plt.xlabel('Predicted label')
  plt.show()

p_test = model.predict(x_test).argmax(axis=1)
cm = confusion_matrix(y_test, p_test)
plot_confusion_matrix(cm, list(range(10)))
```

```
x = MaxPooling2D((2, 2))(x)

x = Conv2D(128, (3, 3), activation='relu',
padding='same')(x)
x = BatchNormalization()(x)
x = Conv2D(128, (3, 3), activation='relu',
padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)

x = Flatten()(x)
x = Dropout(0.2)(x)
x = Dense(1024, activation='relu')(x)
x = Dropout(0.2)(x)
x = Dense(K, activation='softmax')(x)

model = Model(i, x)
```

Compile & Fit

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
r = model.fit(x_train, y_train,
validation_data=(x_test, y_test), epochs=50)
```

Data Augmentation

```
batch_size = 32
data_generator =
tf.keras.preprocessing.image.ImageDataGenerator(
width_shift_range=0.1, height_shift_range=0.1,
horizontal_flip=True)
train_generator = data_generator.flow(x_train,
y_train, batch_size)
steps_per_epoch = x_train.shape[0] // batch_size
r = model.fit_generator(train_generator,
validation_data=(x_test, y_test),
steps_per_epoch=steps_per_epoch, epochs=50)

# lower training accuracy, but higher validation
accuracy, so less overfitting
```

Labels & Misclassified Examples

```
labels = '''airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck'''.split()

misclassified_idx = np.where(p_test != y_test)[0]
i = np.random.choice(misclassified_idx)
plt.imshow(x_test[i], cmap='gray')
plt.title("True label: %s Predicted: %s" %
(labels[y_test[i]], labels[p_test[i]]));
```

Summary

```
model.summary()
```

Chapter 5: Sequenced Data & RNNs

Chapter 5.1: Introduction to Sequenced Data in RNNs

Sequence data refers to things like text, speech, financial data / stock returns. The most popular type is the **time series sequence**, which is any continuous-valued measurement taken periodically, like a company's stock price, or predicting airplane customers over time, or weather tracking, or speech recognition (the Audacity mp3 files are time series of sound amplitudes), or text ("bag-of-words" feature).

For sequenced data, we shall represent the shape of our 3D data array is **N x T x D** as:

- N = # samples
- D = # features
- T = sequence length / time steps in the sequence

Q: Why N x T x D, and not N x D x T? No particular reason except it is the industry standard. Convention is to put features last.

Example 1

For example, if we want to model the path employees take to get to work using GPS data, then:

- N = one sample would be one person's single trip to work. So if there are multiple trips from the same person, those would be separate samples.
- D = 2, as GPS will record latitude and longitude pairs
- T = no. of (lat, lng) measurements taken from start to finish of a single trip. For e.g., if a trip is 30 minutes and coordinates are measured every second, then T = 30 x 60 = 1800.

Then one may ask: wouldn't each person take a different amount of time to get to work? This is true! But in TF / Keras, we deal **only with equal-length sequences** for now.

Q: What if we have variable length sequences, since RNNs are supposed to be equal-length? It is actually possible to use RNNs to handle these, but it is very complicated to work with and are inefficient data structures. One method would be to use single arrays with Numpy (but cannot handle cases where T depends on the sample), or if T depends on sample, we can use T(n) instead where it is the length of n'th sample, so we use a list (inefficient).

Another method would be to pad all samples with 0s until they are all equal length. The drawback is that those really short samples will still take up a lot of memory and processing time despite being short. But it does make coding easier.

Example 2: Single Stock Prices

- D = 1, as stock price is just a single value. D > 1 where we collect more measurements across time.
- N = # of **windows** in the time series, aka assume if we use a window of size T = 10 to predict the next value. This is similar to convolution arithmetic, where if we have a sequence of 100 stock prices, then how many windows of length 10 are there? The answer is 100 – 10 + 1 = 91, where if sequence of length L and window size T, then there are L – T + 1.
- T = 10 as mentioned.

Example 3: Multiple Stocks Prices

Assume if measure stock price every day for 500 different stocks, like S&P500:

- D = 500
- T = 10, arbitrary to be set.
- N = single sample of size T x D = 10 x 500 = 5000.

Example 4: Neuralink

Assume electrodes are connected to the brain to predict what letter you will type, using 1 second of measurements. There is a sampling rate of 1 sample / millisecond (sample here not the same "sample" as N samples of (input, target) pairs, it is just a signal processing terminology).

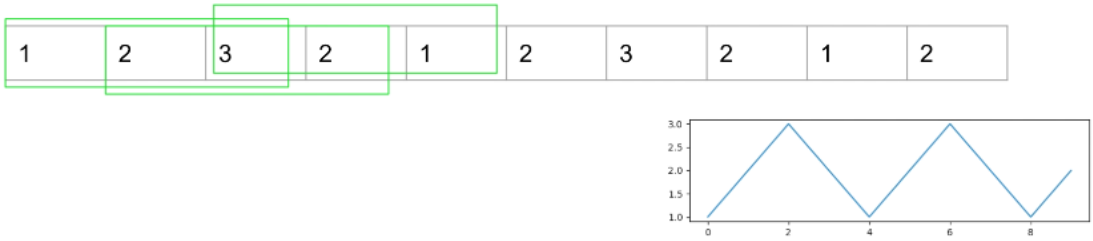
- T = 1000
- N = # of letters test subjects tried to type
- D = # of electrodes

Chapter 5.2: Forecasting

Forecasting here refers to predicting the next value**s** of a time series, AKA predict **multiple** values. For example, you may want to predict demand for 3 – 5 days for products manufactured in a factory. You can also predict the hourly weather for 7 days (7 x 24 = 168 hours = predict 168 steps). The length of time that you want to predict (the number of future steps) is called the **horizon**.

Baby Steps: Linear Regression on 1D Forecasting Time Series

We can flatten our NxTxD data to NxT since D = 1 (only time is a feature). Assume we have a time series of length 10 and we want to predict the next value using the past 3 values. Hence, the input matrix (X) will be of shape Nx3, and the target (Y) will be of shape N.



In this case, N will not be 10 – 3 + 1 = 8, since we cannot use the last window as we don't know the next actual value to compare to the predicted value, AKA we cannot use X8-X9-X10 since we don't have X11. In fact, we have to think differently since we are actually looking at windows of length 4! Hence, N = 7.

x1	x2	x3	x4
x2	x3	x4	x5
x3	x4	x5	x6
x4	x5	x6	x7
x5	x6	x7	x8
x6	x7	x8	x9
x7	x8	x9	x10

```
x = [1, 2, 3, 4, 5, 6]
print(x)

x = np.array(x)
print(x)

x = x.reshape(-1, 2)
print(x)
```

```
[1, 2, 3, 4, 5, 6]
[1 2 3 4 5 6]
[[1 2]
 [3 4]
 [5 6]]
```

```
print(x)
print(x[:-5])
print(x[-5:])
```

```
[0.05042269 0.14987721 0.24783421 0.34331493 0.43536536 0.52306577
 0.60553987 0.68196362 0.75157342 0.81367374]
[0.05042269 0.14987721 0.24783421 0.34331493 0.43536536]
[0.52306577 0.60553987 0.68196362 0.75157342 0.81367374]
```

Therefore, the model will be an **autoregressive (AR) model** in statistics literature. Auto means self, so it is a model that tries to predict the value in a time series using its own values.

$$\hat{x}_t = w_0 + w_1 x_{t-1} + w_2 x_{t-2} + w_3 x_{t-3}$$

The correct way to predict multiple steps into the future is to use our own earlier predictions as input:

$$\hat{x}_4 = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3$$
$$\hat{x}_5 = w_0 + w_1 x_2 + w_2 x_3 + w_3 \hat{x}_4$$
$$\hat{x}_6 = w_0 + w_1 x_3 + w_2 \hat{x}_4 + w_3 \hat{x}_5$$

Therefore, we cannot just use model.predict() in one step. We have to create a loop and call the model to predict only on the most recently generated sequence.

And since "all ML interfaces are the same", we can apply our AR model (linear regression) to an RNN one.

Libraries

```
%tensorflow_version 2.x

import tensorflow as tf
print(tf.__version__)

from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import SGD, Adam

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Data

```
series = np.sin(0.1 * np.arange(200)) +
np.random.randn(200)*0.1 # np.arange() creates range
from 0 to x

plt.plot(series)
plt.show()

# Build Dataset
# We will try to use T past values to predict the next
value.

T = 10
X = []
Y = []

# Rmember that the final target index = len(series) -
1, since Python is zero-indexed.
# Hence, the final t = len(series) - T - 1.

# Assume series = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
# Then index is = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 .
# Then len(series) = 10, T = 3.
# Then it is for range 0, 1, 2, 3, 4, 5, 6.
# For x (features), we will append from 0th to 3rd
index, 1st to 4th index... up to 6th to 9th index.
# For x (features), we will append from 1 to 4, then 2
to 5... up till 7 to 10.
# For y (targets), we will append from 4, 5, up till
10.
```

Modelling

```
i = Input(shape=(T,))
x = Dense(1)(i)
model = Model(i, x)
model.compile(
    loss = 'mse',
    optimizer = Adam(lr=0.1)
)

# Train Model
# Note that for sequential data, we do not do train-
test-split because we need the sequence!
# So instead, we will split data half half.

r = model.fit(
    X[:-N//2], Y[:-N//2],
    epochs=80,
    validation_data=(X[-N//2:], Y[-N//2:])
)
```

Plot Loss per Iteration

```
plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
```

Forecasting

```
# remember that this correct way uses only predicted
values to predict more stuff

validation_target = Y[-N//2:]
validation_predictions = []

# Last Train Input
last_x = X[-N//2] # 1D array of length T

while len(validation_predictions) <
len(validation_target):
  p = model.predict(last_x.reshape(1, -1))[0,0] # 1x1
array -> scalar

  validation_predictions.append(p) # update
predictions list
```

```
for t in range(len(series) - T):
  x = series[t:t+T] # list of x
  X.append(x)
  y = series[t+T] # list of y
  Y.append(y)

X = np.array(X).reshape(-1, T) # make them be 2D
vectors. "-1" means we dunno how many rows, so let
NumPy figure out.
Y = np.array(Y) # 1D vector
N = len(X)

print("X.shape:", X.shape, "Y.shape:", Y.shape, N)
```

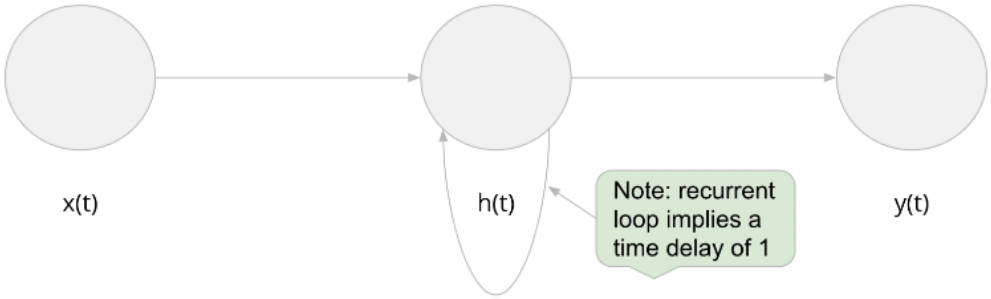
```
  # make new input
  last_x = np.roll(last_x, -1) # shift all elements
in last_x by one to left
  last_x[-1] = p # replace last index no. with new
predicted value

See Results

plt.plot(validation_target, label='forecast target')
plt.plot(validation_predictions, label='forecast
prediction')
plt.legend()
```

Chapter 5.3: Introduction to RNN

Unlike ANNs, we **cannot** flatten the time series data TxD into a single feature vector. This is because a full matrix multiplication will take up too much space (if D = 100, T = 10 000, then you are feeding TxD = 1M sized vector into the feedforward NN). In fact like CNNs, we **don't** have to do this flattening, as RNNs take advantage of the structure.

Make it an RNN

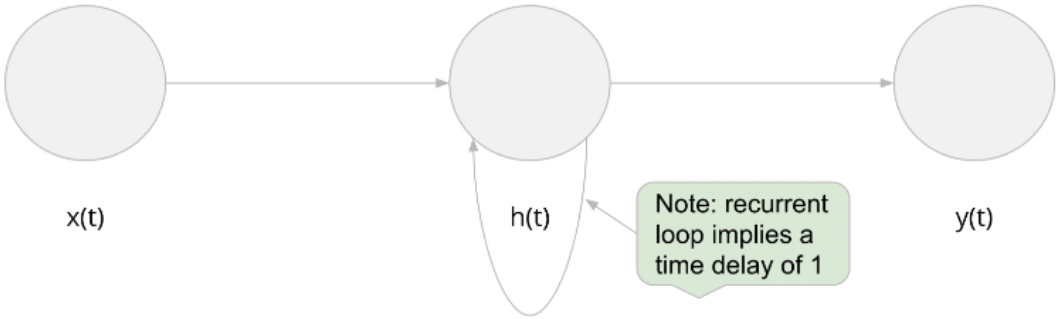
- Make the hidden feature("hidden state") depend on previous hidden state
- Linear regression forecasting model: output is linear function of inputs
- Now: hidden state is a nonlinear function of input and past hidden state
  - What kind of nonlinear function? A neuron!



RNN Equation

- Note: Typically use 1 hidden layer (not 100 like a CNN)
- h(t) is a **nonlinear** function of h(t-1) and x(t)

$$h_t = \sigma(W_{xh}{}^T x_t + W_{hh}{}^T h_{t-1} + b_h)$$
$$\hat{y}_t = \sigma(W_o{}^T h_t + b_o)$$



Helpful Subscripts

- x = input
- h = hidden
- o = output
- xh = input-to-hidden
- hh = hidden-to-hidden

Called the:
"Simple Recurrent Unit"
"Elman Unit"

Sometimes these might just be called W_x and W_h for short

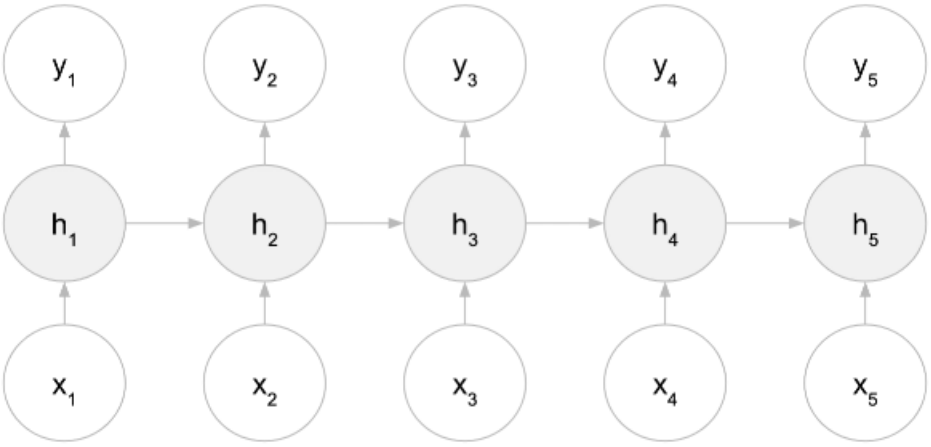
ANN

$$h = \sigma(W_h{}^T x + b_h)$$
$$\hat{y} = \sigma(W_o{}^T h + b_o)$$

RNN

$$h_t = \sigma(W_{xh}{}^T x_t + W_{hh}{}^T h_{t-1} + b_h)$$
$$\hat{y}_t = \sigma(W_o{}^T h_t + b_o)$$

Unrolled RNN



The Mathematical Process

1) First, our data will is a matrix of size TxD. We will then use the formula to calculate the first hidden state h_1 using x_1 and h_0. In the formula below, you will see h_0 is in most of the cases just an array of zeroes. If not, this h_0 is learnable using gradient descent, but usually we stick to just zeroes. There are exceptions but not now.
2) With h_1, we will calculate $\hat{y}_1$.
3) Next, we will load in x_2 and calculate h_2 with x_2 and h_1. With this new h_2, we will calculate $\hat{y}_2$.
4) Repeat the above.

How do we calculate?

- How do we get the output prediction?

h0 is the initial hidden state

It *can* be a learned parameter

In Tensorflow, it's not learnable - just assume it's zero

$$Given: \ x_1, x_2, ..., x_T$$
$$Shape(x_t) = D$$
$$First\ step: \ h_1 = \sigma(W_{xh}{}^T x_1 + W_{hh}{}^T h_0 + b_h)$$
$$What\ is\ h_0?$$

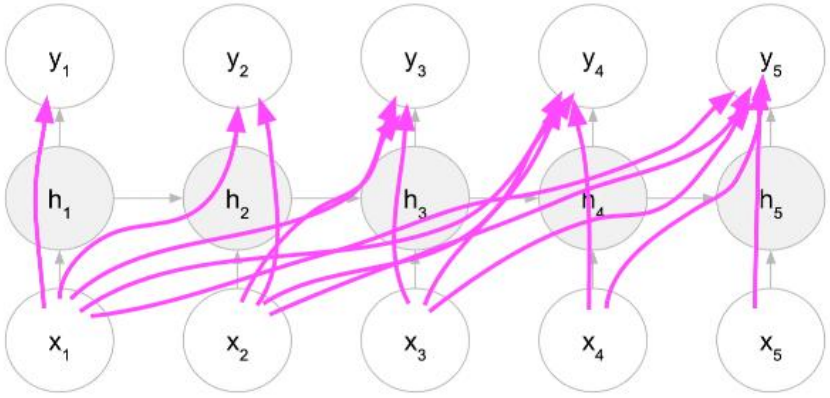
$$We\ have: \ h_1$$
$$Now: \ \hat{y}_1 = \sigma(W_o{}^T h_1 + b_o)$$

$$We\ have:\ x_2,\ h_1 \qquad h_3 = \ldots,\ \hat{y}_3 = \ldots$$

$$h_2 = \sigma(W_{xh}{}^T x_2 + W_{hh}{}^T h_1 + b_h) \ldots$$

$$\hat{y}_2 = \sigma(W_o{}^T h_2 + b_o) \qquad h_T = \ldots,\ \hat{y}_T = \ldots$$

Q: Why need a ŷ for each timestep? Don't we just need one answer? The answer is that we will simply discard the previous outputs / intermediate ŷ-s, and keep only the final $\hat{y}_T$. But this isn't always the case – sometimes we want to keep all the ŷ-s such as in neural machine translation.



$$p(y_1 = k \mid x_1)$$
$$p(y_2 = k \mid x_1,\ x_2)$$
$$p(y_3 = k \mid x_1,\ x_2,\ x_3)$$
$$\ldots$$
$$p(y_T = k \mid x_1,\ x_2,\ x_3,\ \ldots,\ x_T)$$

As you can see from above, y_T depends on x_1 to x_T indirectly as well. RNNs are in contrast to Markov problems, where it is stated that the next word depends only on the immediate previous word and not those before that. That is highly unrealistic, so Markov models are weak in that sense. An RNN will forecast the next word based on <u>all</u> the previous words which is much more powerful.

Relationship to Markov Models

- The Markov assumption is that the current value depends *only* on the immediate previous value
- Easiest to understand with words
- If I tell you the current word is "the", can you *forecast* the next word?

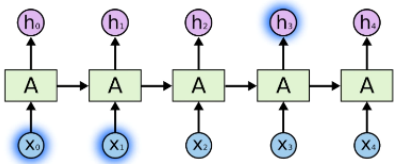
An RNN Language Model

- An RNN will forecast the next word based on *all* the previous words - much more powerful!
- "The quick brown fox jumps over the lazy" → "dog"

$$p(x_{t+1} \mid x_t,\ x_{t-1},\ x_{t-2},\ \ldots,\ x_2,\ x_1)$$

<u>Savings in RNN</u>

- CNNs have "shared weights" to take advantage of structure, resulting in savings - let's see how RNNs do the same
- We use the same W_{xh} for each x(t), and the same W_{hh} to get from h(t-1) to h(t)
- For simplicity's sake, don't consider bias terms



- T = 100
- D = 10
- M = 15 (note: this is still a hyperparameter)
- Flattened input vector: T x D = 1000
- We will have T hidden states: T x M = 1500
- Assume binary classification: K = 1
- Input-to-hidden weight: 1000 x 1500 = 1.5 million
- Hidden-to-output weight: 1500
- Total: ~1.5 million

- W_{xh} - D x M = 10 x 15 = 150
- W_{hh} - M x M = 15 x 15 = 225
- W_o - M x K = 15 x 1 = 15
- Total: 150 + 225 + 15 = 390
- Savings: 1,501,500 / 390 = 3850

Simple RNN Code

Libraries

```
%tensorflow_version 2.x

import tensorflow as tf
print(tf.__version__)

from tensorflow.keras.layers import Input, SimpleRNN,
Dense, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import SGD, Adam

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Data

```
series = np.sin(0.1 * np.arange(200)) +
np.random.randn(200) * 0.1

plt.plot(series)
plt.show()

# Use T past values to predict next value.

T = 10
D = 1
X = []
Y = []

for t in range(len(series) - T):
  x = series[t: t+T]
  X.append(x)
  y = series[t+T]
  Y.append(y)

# Remember that if shape(a,b) = (row, col), then
shape(a,b,c) = (#, row, col).

X = np.array(X).reshape(-1, T, 1) # now data is NxTxD
Y = np.array(Y)
N = len(X)
print("X.shape", X.shape, "Y.shape", Y.shape)
```

Model

```
i = Input(shape=(T, 1))
x = SimpleRNN(5, activation = 'relu')(i) # default
activation is tanh
x = Dense(1)(x)
model = Model(i, x)

model.compile(
    loss = 'mse',
    optimizer = Adam(lr=0.1)
)

r = model.fit(
    X[:-N//2], Y[:N//2],
    epochs=80,
    validation_data=(X[-N//2:], Y[-N//2:])
)
```

Loss per Iteration

```
plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
```

Forecast

```
validation_target = Y[-N//2:]
validation_predictions = []

last_x = X[-N//2] # 1D array of length T

while len(validation_predictions) <
len(validation_target):
  p = model.predict(last_x.reshape(1, -1, 1))[0, 0]

  validation_predictions.append(p)

  last_x = np.roll(last_x, -1)
  last_x[-1] = p

plt.plot(validation_target, label = 'forecast
target')
plt.plot(validation_predictions, label = 'forecast
prediction')
plt.legend()

# An RNN performs worse than an ANN because it is too
customisable and general.
# However, it can do more complex things that an ANN
could not.
# Balance and tradeoffs!

# If you tried using activation=None for the non-
noise data, it'll be perfect because an RNN without
activation is just a linear model, which perfectly
fits the data!
# In contrast, tanh is very inconsistent - can be
super good can be super bad. For relu, it is bad as
well.
```

Shapes in RNNs

Libraries

```
%tensorflow_version 2.x

import tensorflow as tf
print(tf.__version__)

from tensorflow.keras.layers import Input, SimpleRNN,
Dense, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import SGD, Adam
```

Predict

```
Yhat = model.predict(X)
print(Yhat) # shape 1x2 or 1xK
```

Result & Checking

```
model.summary()

a, b, c = model.layers[1].get_weights()
print(a, b, c)
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Data

```
# N = no. of samples
# T = sequence length
# D = no. of input features
# M = no. of hidden nodes
# K = no. of output nodes

N = 1
T = 10
D = 3
K = 2
X = np.random.randn(N, T, D)
print(X)
```

Model

```
M = 5
i = Input(shape=(T, D))
x = SimpleRNN(M)(i)
x = Dense(K)(x)

model = Model(i, x)
print(a.shape, b.shape, c.shape)

# Wxh is DxM, Whh is MxM, bias is M

Wxh, Whh, bh = model.layers[1].get_weights()
Wo, bo = model.layers[2].get_weights()
```

Manual Math

```
h_last = np.zeros(M) # initial hidden state
x = X[0] # the one and only sample
Yhats = [] # to store

for t in range(T):
  h = np.tanh(x[t].dot(Wxh) + h_last.dot(Whh) + bh) #
TxD.DxM + 1xM.MxM + 1xM
  y = h.dot(Wo) + bo # only care about this value on
the last iteration. TxM.MxK + 1xK
  Yhats.append(y)

  h_last = h # assign h to h_last

print(Yhats[-1]) # Yhats is TxK, but we just want the
last one
```

Chapter 5.4: GRU & LSTM

These are modern RNN units called **Gated Recurrent Unit (GRU)** and **Long Short-Term Memory (LSTM)**. They are essentially the same thing, with GRUs being a simplified version of LSTM (less parameters and thus more efficient).

Why Do We Need Them?

Don't we already have the Simple RNN already? To answer this, we need to consider the vanishing gradient problem.

The output prediction is a huge composite function of x_1, x_2, …, x_T. W_{xh} appears several times as well, once at each time step. In fact, W_{xh} will be transposed and then multiply with x_t for t = 1…T. Hence, we need to find the gradient of $W_{xh}^T x_t$ for all t = 1…T. The final gradient will be a function of all these individual gradients:

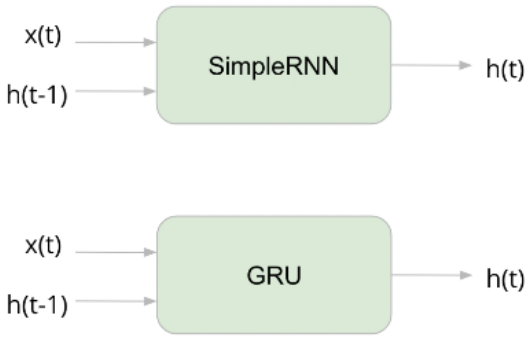
$$\frac{\partial J}{\partial W_{xh}} = f\left(\frac{\partial (W_{xh}{}^T x_T)}{\partial W_{xh}}, \frac{\partial (W_{xh}{}^T x_{T-1})}{\partial W_{xh}}, \cdots, \frac{\partial (W_{xh}{}^T x_1)}{\partial W_{xh}}\right)$$

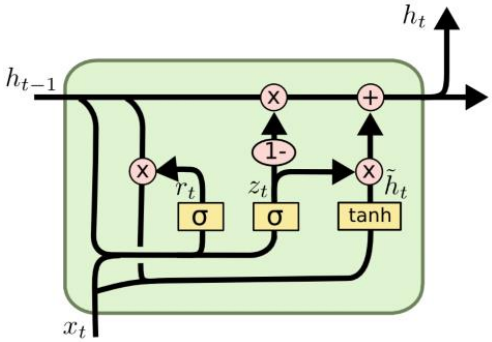
$$\hat{y}(T) = \sigma(W_o{}^T \sigma(W_{xh}{}^T x_T + W_{hh}{}^T \sigma(W_{xh}{}^T x_{T-1} + W_{hh}{}^T \sigma(\ldots(W_{xh}{}^T x_1 + W_{hh}{}^T h_0 \ldots)\ldots))))$$

Note how that x_1 is the most deeply nested term in the equation, followed by x_2 and so on. Since ANN is a big composite function – composite functions turn into multiplications in the derivative / gradient (chain rule). Therefore, the more deeply nested you are, the more multiplications. In other words, **RNNs are vulnerable to the vanishing gradient problem** – the farther back in input x_t is, the more its gradient vanishes. Hence, the simple RNN cannot learn from inputs too far back.

An example would be in NLP. If you asked the simple RNN: "what day was Albert Einstein born?". By the time the RNN has "read" the whole article, it will forget what was the beginning! It would have forgotten the earlier parts of the article. This is where GRUs and LSTM will come in to save the day.

5.4.1: Gated Recurrent Unit (GRU)





The **GRU** has a similar API (first picture) as a Simple RNN, just that the calcuations are different.

Furthermore, the middle picture is something that may help you understand it.

Mathematical Equations

Simple RNN unit, for comparison

$$h_t = \tanh(W_{xh}{}^T x_t + W_{hh}{}^T h_{t-1} + b_h)$$

z(t) = update gate vector
r(t) = reset gate vector
h(t) = hidden state

$$z_t = \sigma(W_{xz}{}^T x_t + W_{hz}{}^T h_{t-1} + b_z)$$
$$r_t = \sigma(W_{xr}{}^T x_t + W_{hr}{}^T h_{t-1} + b_r)$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tanh(W_{xh}{}^T x_t + W_{hh}{}^T (r_t \odot h_{t-1}) + b_h)$$

Unlike the Simple RNN composed of just one formula, the GRU uses three formulas:

- All z(t), r(t), h(t) are all vectors of size M. M refers to the no. of hidden units / features.
- Any weight going from x(t) is DxM. Any weight going from h(t) is MxM. All bias terms are of size M.
- Note that the circle with a dot picture is an **element-wise multiplication**. This means that $(1 - z_t)$ is an M-sized vector, and that h_{t-1} is an M-sized vector as well. Both will multiply as per element.

Let us break down the formulas:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tanh(W_{xh}{}^T x_t + W_{hh}{}^T (r_t \odot h_{t-1}) + b_h)$$

$$h_t = p(keep\ h_{t-1}) h_{t-1} + p(discard\ h_{t-1}) SimpleRNN(x_t,\ h_{t-1})$$

- z_t: "When in the third equation, should I take the new value for h(t), or keep the old value of h(t-1)?"
  - The vanishing gradient means that RNN forgets things in the past.
  - Since z_t uses sigmoid, it is between 0 and 1. If it approaches 1, then h(t-1) is forgotten for a new h(t).
  - Now, we can explicitly remember the previous h(t-1) if z(t) is small and approaches zero.
  - Remembers the previous hidden state to carry forward to the next hidden state.
  - The equation behaves just like a logistic regression / a neuron, telling us to choose h(t) or h(t-1).

$$r_t = \sigma(W_{xr}{}^T x_t + W_{hr}{}^T h_{t-1} + b_r)$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tanh(W_{xh}{}^T x_t + W_{hh}{}^T (r_t \odot h_{t-1}) + b_h)$$

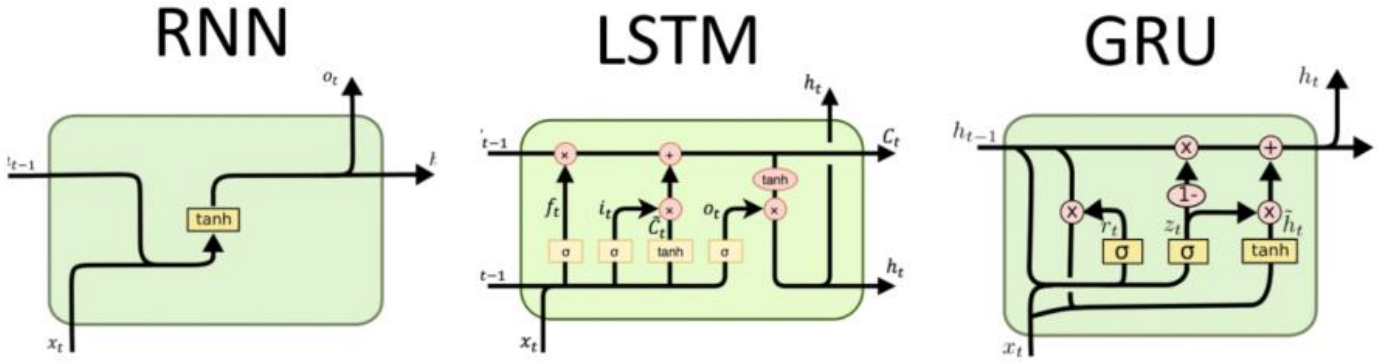
- r_t is just another switch to remember or forget h(t-1) in the so-called SimpleRNN side.
    - Values from sigmoid so it is between 0 and 1.
    - It does an element-wise multiplication with h(t-1).
    - Hence if r_t is composed of values close to zero, then the output of the element-wise multiplication will be close to zero as well, forgetting the values. If r_t is close to 1, then the values will stay almost the same, remembering the values.
    - Therefore, it is used to decide what values it wants to remember and forget, before multiplying W_{hh}.
    - Its function is very similar to z_t, both just help to remember or forget different h(t-1)s.

Summary

- Same API as Simple RNN.
- Output is h(t), which values depend on h(t-1) and x(t).
- The difference is that it has gates to remember or forget each component of h(t-1).
- Simple RNNs have no choice but to eventually forget things it saw earlier in a sequence, due to the vanishing gradient problem. GRUs instead use binary classifiers (logistic regression neurons) as our gates to choose to remember or forget old values.

5.4.2: LSTM

Modern experiments have shown and proven that LSTMs are better than GRUs. So use LSTMs. LSTMs are like GRUs but with more state vectors and more gates.



As seen below, the LSTM is similar to the GRU except that it has an additional state known as the **cell state c(t)**. But it is usually ignored as it just helps to calculate the intermediate values only. You can ignore it as well.

GRU vs LSTM

- Not exactly the same API
- LSTM returns 2 states:
    - Hidden state h(t)
    - Cell state c(t) (usually ignored)
    - LSTM unit in Tensorflow outputs h(T), but can also optionally output c(T)
    - Also means you need 2 initial states: c_0 and h_0

Mathematical Equations

$$f_t = \sigma(W_{xf}{}^T x_t + W_{hf}{}^T h_{t-1} + b_f)$$
$$i_t = \sigma(W_{xi}{}^T x_t + W_{hi}{}^T h_{t-1} + b_i)$$
$$o_t = \sigma(W_{xo}{}^T x_t + W_{ho}{}^T h_{t-1} + b_o)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot f_c(W_{xc}{}^T x_t + W_{hc}{}^T h_{t-1} + b_c)$$
$$h_t = o_t \odot f_h(c_t)$$

Forget gate

Input/update gate

Output gate

"Simple RNN"

f(t) = Neuron (binary classifier)

i(t) = Neuron (binary classifier)

o(t) = Neuron (binary classifier)

c(t) = f(t) * c(t-1) + i(t) * SimpleRNN

h(t) = o(t) * tanh(c(t))

Do not be alarmed at these equations. They are merely neurons! They merely do binary classifying!

- f(t) is the **forget gate** that will make c(t) forget its old value or not.
- i(t) is the **input / update gate**, similar to z_t in GRU.
- o(t) is just the **output gate**.
- c(t) is the **cell state**, which has a similar role of h_t in the GRU. Specifically, we have the weighted sum of two terms: the former is controlled by the forget gate on how much of the old values we want to remember or forget; while the latter is a Simple RNN term but without any forget gates inside BUT is controlled by the input gate on how much to remember as well.
- h_t is the **hidden gate**, which is a simple transformation of c(t) and has output gate o_t which controls which values of the cell state we actually pass through to the hidden state h_t.
- f_c and f_h seen in c_t and h_t respectively can be anything, but usually by default **tanh**. We do not change them in TF.

Coding

Options for RNN Units

- For each $x_1, x_2, ..., x_T$, we will calculate $h_1, h_2, ..., h_T$
- We've seen that the SimpleRNN, GRU, and LSTM will return h_T by default
- We may want all of $h_1, h_2, ..., h_T$, to get $\hat{y}_1, \hat{y}_2, ..., \hat{y}_T$

```
i = Input(shape=(T,))
x = LSTM(M, return_sequences=True)(i)
x = Dense(K)(x)
```

N x T x M

N x T x K

Options for RNN Units

- `o, h = SimpleRNN(M, return_state=True)(i)`
- "o" and "h" are the same thing - final state h_T
- `o, h = GRU(M, return_state=True)(i)`
- "o" and "h" are the same thing - final state h_T
- `o, h, c = LSTM(M, return_state=True)(i)`
- "o" and "h" are the same thing - final state h_T, but we also get c_T

ANN vs SimpleRNN vs LSTM (short distance)

Libraries

```
%tensorflow_version 2.x

import tensorflow as tf
print(tf.__version__)

from tensorflow.keras.layers import Input,
SimpleRNN, GRU, LSTM, Dense, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import SGD, Adam

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Data

```
series = np.sin((0.1 * np.arange(400)) ** 2) #
time series of the form x(t) = sin(omega * t^2)
```

Multi-step forecast (AR)

```
validation_target = Y[-N//2:]
validation_predictions = []

# last train input
last_x = X[-N//2] # 1-D array of length T
```

One-Step Forecast using True Targets (RNN/LSTM)

```
outputs = model.predict(X)
print(outputs.shape)
predictions = outputs[:, 0]

plt.plot(Y, label='targets')
plt.plot(predictions, label='predictions')
plt.title("many-to-one RNN")
plt.legend()
plt.show()
```

```
plt.plot(series)
plt.show()

T = 10
D = 1
X = []
Y = []

for t in range(len(series) - T): # do this 190
times
  x = series[t : t+T]
  X.append(x)
  y = series[t+T]
  Y.append(y)

# print(X[:3], "\n", Y[:3])

X = np.array(X).reshape(-1, T) # make it NxT
Y = np.array(Y)

# print(X[:3], "\n", Y[:3])

N = len(X)
print("X.shape", X.shape, "Y.shape", Y.shape, N)
```

ANN Model

```
# Trying ANN Model

i = Input(shape=(T,))
x = Dense(1)(i)
model = Model(i, x)
model.compile(loss='mse', optimizer=Adam(lr=0.01))

r = model.fit(X[:N//2], Y[:-N//2], epochs=80,
validation_data=(X[-N//2:], Y[-N//2:]))
```

SimpleRNN / LSTM Model

```
# Try RNN / LSTM Model

X = X.reshape(-1, T, 1) # make it NxTxD

i = Input(shape=(T, D))
x = LSTM(10)(i) # change to SimpleRNN or LSTM as
you wish
x = Dense(1)(x)
model = Model(i, x)
model.compile(loss='mse', optimizer=Adam(lr=0.05))

r = model.fit(X[:-N//2], Y[:-N//2], batch_size=32,
epochs=200, validation_data=(X[-N//2:], Y[-
N//2:]))
```

Loss per Iteration (Both)

```
plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()

# Conclusion: For a simple sine wave without
noise, a linear model does best while RNN does
terribly with default parameters due to too much
flexibility.
# But now with a noisy signal, a linear model
cannot solve it at all. The RNN has the
flexibility to match the signal.
# When SimpleRNN:
# Good predictions even when signal slows down in
second half. This is despite the fact the model
never saw such a low frequency in the training
set.
# But remember that single

# When LSTM:
# Actually looks quite good.
```

Multi-Step using Predicted Targets (RNN/LSTM)

```
forecast = []
input_ = X[-N//2]

while len(forecast) < len(Y[-N//2:]):
  # Reshape input_ to NxTxD
  f = model.predict(input_.reshape(1, T, 1))[0, 0]
  forecast.append(f)

  # Make a new input with the latest forecast
  input_ = np.roll(input_, -1)
  input_[-1] = f

plt.plot(Y[-N//2:], label='targets')
plt.plot(forecast, label='forecast')
plt.title("RNN Forecast")
plt.legend()
plt.show()

# When SimpleRNN:
# Better than linear model, and only good at the
start. Bad at the ends. It doesn't know that in
the second half the signal slows down so that's
why it failed at the second half.
# I guess one note to make is that RNNs in multi-
step forecast do not do well if there are big
changes.

# When LSTM:
# Good, but cannot capture the middle low
frequency part. So same as the SimpleRNN.
# Q: Why LSTM fail here though? A: Cannot assume
cos LSTM is popular means it's the best. LSTM are
better than RNNs at finding long term
dependencies, but doesn't mean they're better at
everything. In fact, this fact doesn't hold true
for arbitrarily long term dependencies - there are
times when LSTM will 'forget'. This dataset does
not really have long term dependencies also at
least for the timespan of the dataset. Data from
very far in the past isn't going to help predict
future values. There is no real advantage to using
LSTM for this problem.
```

Long Distance w ANN, SimpleRNN, & LSTM

Libraries

```
%tensorflow_version 2.x

import tensorflow as tf
print(tf.__version__)

from tensorflow.keras.layers import Input,
SimpleRNN, GRU, LSTM, Dense, Flatten,
GlobalMaxPool1D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import SGD, Adam

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Data (Nonlinear AND Long-Distance)

```
T = 30 # start small eg 10 and increase it later
D = 1
X = []
Y = []

def get_label(x, i1, i2, i3):
  # x = sequence
  if x[i1] < 0 and x[i2] < 0 and x[i3] < 0:
    return 1
  if x[i1] < 0 and x[i2] > 0 and x[i3] > 0:
    return 1
  if x[i1] > 0 and x[i2] < 0 and x[i3] > 0:
    return 1
  if x[i1] > 0 and x[i2] > 0 and x[i3] < 0:
    return 1
  return 0

# The formula is getting 3 random data points, and
getting their signs (positive or negative). There
are 8 possible combinations as per picture.
# We organise these 3 data points on a 3D axis,
then we can have a 3D version of the XOR. We have
8 quadrants and we will make sure that any two
adjacent quadrants do not belong to the same
class.
# Note the short distance and long distance y. If
we choose the last 3 y's, then the model doesn't
have to remember for so long (it is easy). If we
choose the first 3 y's, then it is harder as the
RNN has to remember the sequence of 3 random
numbers all the way to the end. This gets harder
if the sequence is longer. So here we can prove
that LSTM can remember stuff.

for t in range(5000):
  x = np.random.randn(T) # completely random, so
each data point is not related to the ones beside
it, unlike a time series.
  X.append(x)
  #y = get_label(x, -1, -2, -3) # short distance
  y = get_label(x, 0, 1, 2) # long distance
  Y.append(y)

X = np.array(X)
Y = np.array(Y)
N = len(X)

print(X[:5], Y[:5], N)
```

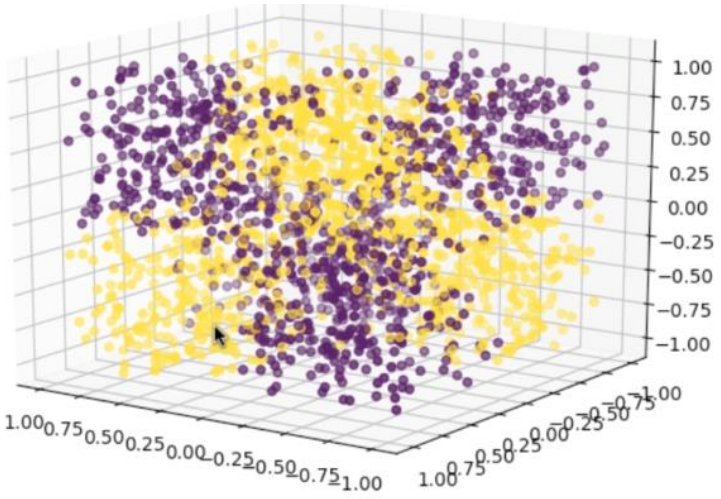
Modelling

```
inputs = np.expand_dims(X, -1)

i = Input(shape=(T, D))
# x = SimpleRNN(5)(i)
# x = GRU(5)(i)
# x = LSTM(5)(i)
x = LSTM(5, return_sequences=True)(i)
x = GlobalMaxPool1D()(x)
x = Dense(1, activation = 'sigmoid')(x)
model = Model(i, x)

model.compile(
    loss='binary_crossentropy',
    #optimizer = 'rmsprop',
    #optimizer = 'adam',
    #optimizer = Adam(lr=0.01),
    #optimizer = SGD(lr=0.1, momentum=0.9),
    metrics = ['accuracy'])

r = model.fit(
    inputs, Y,
    epochs = 100,
    validation_split = 0.5
)
```

Review

```
plt.plot(r.history['loss'], label = 'loss')
plt.plot(r.history['val_loss'], label =
'val_loss')
plt.legend()

plt.plot(r.history['accuracy'], label = 'acc')
plt.plot(r.history['val_accuracy'], label =
'val_acc')
plt.legend()

# When T = 10:
# For short distance + SimpleRNN = very good results.
Cos no need long-term memory + no vanishing gradient
problem.
# For long distance + SimpleRNN = not as good result,
but still quite high actually. But pattern not
consistent. If you run several times, you may find out
it may fail.
# For long distance + LSTM = very good and consistent
results, especially at the end. That middle part though.
# When T = 20:
# For long distance + Simple RNN = failure.
# For long distance + LSTM = success?! The video shows
good but mine not that nice.
# For long distance + GRU = good but only after 400
epochs. GRU weaker than LSTM here.
# When T = 30 and epochs = 400 or 100:
# For long distance + LSTM = only 50% not good. Also
will split far apart. LSTM not good at THAT far back.
# For special LSTM model = very good. Remember that by
default, LSTM only returns the last h(t) as output. But
if return_sequences=True, then you will get all the
hidden states for each timestep. GlobalMaxPool1D
collapses away a single dimension - it is just an
element-wise max() operation of h1 h2... hT. You will
end up with a vector of size M and pass it through the
final dense layer.
# The special LSTM model works despite at 100 epochs
because GMP1D allowed us to handle longer sequences by
looping over all the hidden states, which is much better
than the default LSTM that had to remember everything
you saw previously up until the final hT. It is akin to
searching along every hidden state and picking the most
useful ones.
```

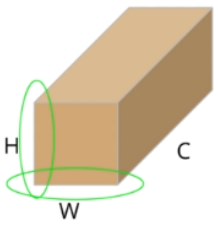
x(1)	x(2)	x(3)
+	+	+
+	+	-
+	-	+
+	-	-
-	+	+
-	+	-
-	-	+
-	-	-

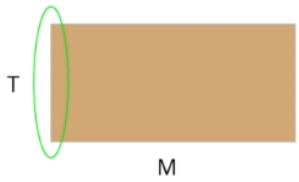


- return_sequences=False
  - Input: T x D
  - After RNN Unit h(T): M
  - Output: K
- return_sequences=True
  - Input: T x D
  - After RNN Unit h(1), h(2), ..., h(T): T x M
  - After global max pooling max{h(1), h(2), ..., h(T)}: M
  - Output: K

Global Max Pooling

- For images, use GlobalMaxPooling2D
  - Input: H x W x C
  - Output: C
- For sequences, use GlobalMaxPooling1D
  - Input: T x M
  - Output: M





Chapter 5.5: Image Classification with RNN

MNIST with LSTM

Libraries

```
%tensorflow_version 2.x
import tensorflow as tf
print(tf.__version__)

from tensorflow.keras.layers import Input,
SimpleRNN, GRU, LSTM, Dense, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import SGD, Adam

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Data

```
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) =
mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
print("x_train.shape:", x_train.shape)
```

Model

```
i = Input(shape=x_train[0].shape)
x = LSTM(128)(i)
x = Dense(10, activation = 'softmax')(x)
model = Model(i, x)

model.compile(optimizer='adam',
              loss =
'sparse_categorical_crossentropy',
              metrics=['accuracy']
)
```

```
r = model.fit(x_train, y_train,
validation_data=(x_test, y_test), epochs=10)
```

Evaluation

```
plt.plot(r.history['loss'], label = 'loss')
plt.plot(r.history['val_loss'], label =
'val_loss')
plt.legend()

plt.plot(r.history['accuracy'], label = 'acc')
plt.plot(r.history['val_accuracy'], label =
'val_acc')
plt.legend()

# We get 99% accuracy on train and test sets,
despite just 10 epochs. The NN appears to
generalise well and captures long distance
information in the image well.
# This is long distance because you cannot know
what digit the images of unless you consider the
image as a whole. You cannot just look at the last
five rows of an image to determine what the images
are.

# SO YES, you can do images with LSTMs.
```

Chapter 5.6 Stock Prediction with LSTM

Stock Predictions with LSTM

Libraries

```
%tensorflow_version 2.x
import tensorflow as tf
print(tf.__version__)

from tensorflow.keras.layers import Input, LSTM, GRU,
SimpleRNN, Dense, GlobalMaxPool1D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import SGD, Adam

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
```

Data

```
df = pd.read_csv('https://raw.githubusercontent.
com/lazyprogrammer/machine_learning_examples/
master/tf2.0/sbux.csv')
df.head() # Starbucks stock price Feb 13 to Feb 18

series = df['close'].values.reshape(-1, 1) # reshape
to Nx1 vector
```

```
for t in range(len(series) - T):
  x = series[t: t+T]
  X.append(x)
  y = series[t+T]
  Y.append(y)

X = np.array(X).reshape(-1, T, 1) # NxTxD
Y = np.array(Y)
N = len(X)

print("X.shape", X.shape, "Y.shape", Y.shape, N)
```

The Model 2

```
# LSTM AR Model

i = Input(shape=(T, 1))
x = LSTM(5)(i)
x = Dense(1)(x)
model = Model(i, x)

model.compile(loss='mse', optimizer=Adam(lr=0.1))

r = model.fit(X[:-N//2], Y[:-N//2], epochs=80,
validation_data=(X[-N//2:], Y[-N//2:]))
```

```
# Normalise the data as some are in millions while
others in twenties.
scaler = StandardScaler()
scaler.fit(series[:len(series)//2]) # only train on
the trainset. Why? Because it follow the principle of:
anything you learn, must be learned from the model's
training data. So you use the trainset's mean and SD
to the whole data.
series = scaler.transform(series).flatten() # as it
was in 2D array, we flatten to a 1D vector

# Use T past values to predict next value.

T = 10
D = 1
X = []
Y = []

for t in range(len(series) - T):
  x = series[t: t+T]
  X.append(x)
  y = series[t+T]
  Y.append(y)

X = np.array(X).reshape(-1, T, 1) # NxTxD
Y = np.array(Y)
N = len(X)

print("X.shape", X.shape, "Y.shape", Y.shape, N)
```

Model 1: LSTM AR Model

```
# LSTM AR Model

i = Input(shape=(T, 1))
x = LSTM(5)(i)
x = Dense(1)(x)
model = Model(i, x)

model.compile(loss='mse', optimizer=Adam(lr=0.1))

r = model.fit(X[:-N//2], Y[:-N//2], epochs=80,
validation_data=(X[-N//2:], Y[-N//2:]))

plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()

# Loss does decrease, so LSTM can predict somewhat.

# One Step Forecast using True Targets (wrong method)
outputs = model.predict(X)
print(outputs.shape)
print(outputs)
predictions = outputs[:, 0]

plt.plot(Y, label='targets')
plt.plot(predictions, label='predictions')
plt.legend()
plt.show()

# Really amazing results, but one-step forecasts are
not really helpful.

# Multiple Step Forecasts
validation_target = Y[-N//2:] # actual
validation_predictions = [] # predictions

last_x = X[-N//2] # 1D array of length T

plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()

# Model has a harder time learning anything. Loss goes
down but val_loss goes up means we are just fitting to
noise.

# One Step Forecast using True Targets (wrong method)
outputs = model.predict(X)
print(outputs.shape)
predictions = outputs[:, 0]

plt.plot(Y, label='targets')
plt.plot(predictions, label='predictions')
plt.legend()
plt.show()

# Dunno if good results, but judging by loss results,
probably not good.

# Multiple Step Forecasts
validation_target = Y[-N//2:] # actual
validation_predictions = [] # predictions

last_x = X[-N//2] # 1D array of length T

while len(validation_predictions) <
len(validation_target):
  p = model.predict(last_x.reshape(1, T, 1))[0, 0] #
1x1 array -> scalar

  validation_predictions.append(p)

  last_x = np.roll(last_x, -1)
  last_x[-1] = p

plt.plot(validation_target, label='forecast target')
plt.plot(validation_predictions, label='forecast
prediction')
plt.legend()

# Not good.
```

Model 3: All Columns LSTM

```
# The third model will instead use all columns (D=5)
and just predict if price goes up or down.

input_data = df[['open', 'high', 'low', 'close',
'volume']].values
targets = df['Return'].values

T = 10 # no. of time steps to look at to predict next
day
D = input_data.shape[1]
N = len(input_data) - T # (e.g. if T=10 and you have 11
data points then you'd only have 1 sample)

# Normalise the Inputs

Ntrain = len(input_data) * 2 // 3
scaler = StandardScaler()
scaler.fit(input_data[:Ntrain + T])
input_data = scaler.transform(input_data)

# Other Set Ups

X_train = np.zeros((Ntrain, T, D))
Y_train = np.zeros(Ntrain)
```

```
while len(validation_predictions) <
len(validation_target):
  p = model.predict(last_x.reshape(1, T, 1))[0, 0] #
1x1 array -> scalar

  validation_predictions.append(p)

  last_x = np.roll(last_x, -1)
  last_x[-1] = p

plt.plot(validation_target, label='forecast target')
plt.plot(validation_predictions, label='forecast
prediction')
plt.legend()

# Straight line. Means the model isn't predicting the
next value in the time series, it's just copying the
previous value.
# This is to be expected, because in time series, the
data is highly correlated - neighbouring data are
close, eg if one pixel is red the next should be red,
eg if time then it's a smooth function and won't jump
around.

# Conclusion: It is misleading and unconventional to
use one-step predictions.
# What is more conventional is the STOCK RETURN. That
is R = (Vfinal - Vinitial) / Vinitial.
```

Model 2: Returns with LSTM

```
# Calculate Returns by first shifting the data.
Yesterday's closing price is aligned with today's
closing price.
df['PrevClose'] = df['close'].shift(1) # move
everything down 1

# so it's now like
# close / prev close
# x[2] x[1]
# x[3] x[2]
# x[4] x[3]
# ...
# x[t] x[t-1]

df.head()

# Then the return formula:
df['Return'] = (df['close'] - df['PrevClose']) /
df['PrevClose']
df.head()

df['Return'].hist()

# The histogram showed us pretty small values, so we
want to normalise them.

series = df['Return'].values[1:].reshape(-1, 1)

scaler = StandardScaler()
scaler.fit(series[:len(series) // 2])
series = scaler.transform(series).flatten()

# Use T past values to predict next value.

T = 10
D = 1
X = []
Y = []
for t in range(Ntrain):
  X_train[t, :, :] = input_data[t: t+T]
  Y_train[t] = (targets[t+T] > 0)

X_test = np.zeros((N - Ntrain, T, D))
Y_test = np.zeros(N - Ntrain)

for u in range(N - Ntrain):
  # u counts from 0...(N - Ntrain)
  # t counts from Ntrain...N
  t = u + Ntrain
  X_test[u, :, :] = input_data[t:t+T]
  Y_test[u] = (targets[t+T] > 0)
```

Model 3: LSTM

```
i = Input(shape=(T, D))
x = LSTM(50)(i)
x = Dense(1, activation='sigmoid')(x)
model = Model(i, x)
model.compile(
    loss = 'binary_crossentropy',
    optimizer = Adam(lr=0.001),
    metrics = ['accuracy']
)

r = model.fit(
    X_train, Y_train,
    batch_size = 32,
    epochs = 300,
    validation_data = (X_test, Y_test)
)

plt.plot(r.history['loss'], label = 'loss')
plt.plot(r.history['val_loss'], label = 'val_loss')
plt.legend()
plt.show()

# Train loss down but val loss up = model is
overfitting to the noise in the train set.

plt.plot(r.history['accuracy'], label = 'acc')
plt.plot(r.history['val_accuracy'], label = 'val_acc')
plt.legend()
plt.show()

# Same. Train acc goes up but val acc stays constant.
# Q: Why is val_acc constant at 50%? Note that for
binary classification, the worst accuracy is NOT 0%,
but rather 50%! It means it is no better than random
guessing. If val_acc is 0%, you can just reverse all
your predictions and you will get 100% accuracy for a
perfect model!
# Hence we had the worst model.
```

Meta-Conclusion

```
# If Model 3 which is just to predict up/down fails,
then Model 1 and 2 are doomed to fail as well. If
cannot even predict up/down, how to predict stock price
or returns?
# If you look back, we had problems even predicting a
sine wave (a deterministic formula which should be
easy).
# The problem with predicting stock prices from
historical prices is that you are ignoring lots of
real-world events.
# LSTMs are not good at predicting stock prices. But it
is good at language modelling, machine translation, and
even image classification!
```

Chapter 6: Natural Language Processing (NLP)

Chapter 6.1: Introduction to NLP & Embeddings

Text is a sequence data, but it is not continuous – it is categorical. If an input is ["the", "quick", "brown", "fox"…], then x_1 is "the", x_2 is "quick" etc. Since they are words, we cannot use sigmoid or other math with them.

You might think that since they're categories, we can apply one-hot encoding to get the categories. If there are 3 million words in a dictionary, then make a 3Mx3M matrix of mostly zeroes, and a "1" to denote which word it is. But this leads to problems – if a dictionary is large with many possible tokens / words, then your weight matrix would also be very large. Another problem is that if you mapped them all out, since they are just a single 1 and many 0s, then the Euclidean distance between them will always be sqrt(2) – therefore all words are equally apart! This makes it useless for math operations.

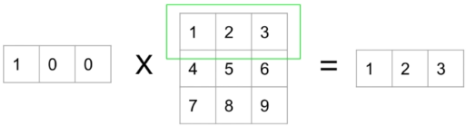
There is a solution, which is to assign each word to a **D-dimensional vector** (not one-hot encoded) using an **embedding layer**.

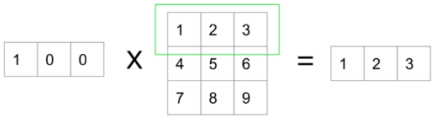
Coding Tricks

Basically, if you want a certain row from the weight matrix, just multiply by 1 at the K^{th} index! Or an even convenient method would be just to index the weight matrix itself!

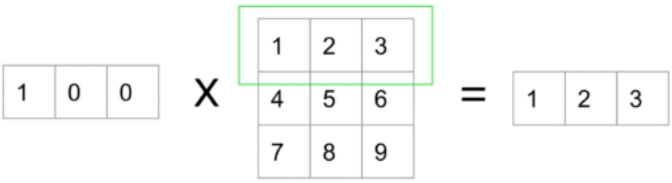
- If the index set to 1 is located at position 1, we get the *first row*
- If the index set to 1 is located at position 2, we get the *second row*
- If the index set to 1 is located at position 3, we get the *third row*

- One-hot encoding an integer k and multiplying that by a matrix is the *same* as selecting the k'th row of the matrix
- E.g one_hot(k) * W == W[k]





- Old way takes 2 steps:
  - Create a vector of size V containing all zeros, set the k'th entry to 1
  - Multiply the one hot vector by the weight matrix
- New way takes 1 step:
  - Index the weight matrix (W[k])

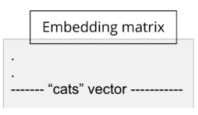


This is exactly what the **Embedding layer** in TF2 does. All you have to do is to map your dataset with sequences of words into a dataset with sequences of unique word indices. Now that you have only integers, you can use an embedding layer to map each word's integer to a corresponding word vector. From then on out, you will have a TxD matrix to insert into your RNN.

Tensorflow Embedding

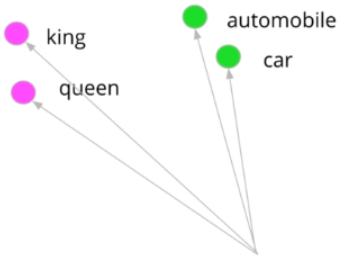
- This is exactly what the Embedding layer does!
- STEP #1: Convert words into integers
  - ["I", "Like", "Cats"] → [50, 25, 3]
- STEP #2: Use integers to index the word embedding matrix to get word vectors for each word
  - [50, 25, 3] → [[0.3, -0.5], [1.2, -0.7], [-2.1, 0.9]]
  - T-length array → T x D matrix

- Convert your sentences into sequences of word indexes:
  - ["I", "Like", "Cats"] → [50, 25, 3]
- Embedding layer maps sequence of integers into sequence of word vectors
  - [50, 25, 3] → [[0.3, -0.5], [1.2, -0.7], [-2.1, 0.9]]
  - T-length array → T x D matrix



How are the weights found?

- We know intuitively that the word vectors must have some useful structure
- Luckily, it's the same story as with convolutional filters
- These weights are found *automatically* with gradient descent when you call `model.fit()`



You may come across **pre-trained word vectors**. But we will not be doing them for this course.

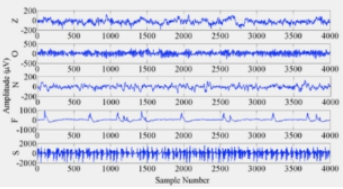
Pre-trained word vectors

- Sometimes, we use *pre-trained* word vectors (trained using some other algorithm)
- We freeze (fix) the embedding layer's weights, so only the *other* layers are trained when we call `model.fit()`
- You're encouraged to read about word2vec and GloVe to learn more

Essentially, an **<u>embedding layer just helps to convert a sequence of words into a TxD matrix of numbers that can be passed into an RNN (or other models)</u>**.

Summary

- It's not possible to multiply a *word* by a weight matrix - there is no concept of multiplying words
- We must turn words into some kind of numerical representation first
- One-hot encoding is a fine first guess
  - Too much space (there could be up to 1 million tokens)
  - Not geometrically useful ("cat" is just as close to "feline" as it is to "airplane")
  - We need to make use of "machine learning is nothing but a geometry problem"
- Better: convert words to unique integers
- Use those integers to *index* a weight matrix (the *embedding matrix*)
- Much faster than matrix multiplication

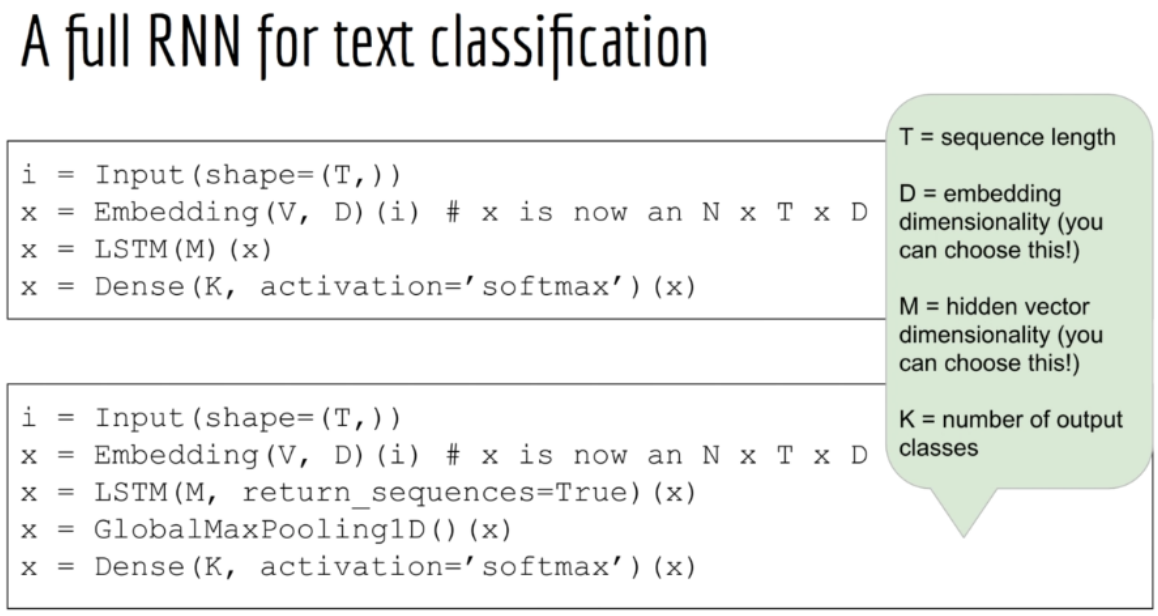


- Each input sentence thus becomes a T x D sequence which we know an RNN can accept
- The embedding is trained like any other layer - we just need to call `model.fit()`
- Read up on word2vec and GloVe for alternative methods for training embeddings

Male-Female Verb tense Country-Capital

Chapter 6.2: Code Preparation (NLP)

- Data is a list of strings, usually called a **"document"**.
- Since words -> integers -> vectors, hence we need a **mapping** from word -> integer.
- Take note that when mapping words, **do not** use "0". That "0" is reserved for padding empty values.
- When we read a file, the text is read as strings. Hence, we need to break it up into individual single words, in a process called **tokenisation**. Therefore: string -> tokens -> integers -> vectors.
- TF2 has a **Tokenizer** that helps to convert strings into tokens, and subsequently into integers.
- TF2 also has a **pad_sequences()** function that helps to pad sequences with 0s. You can **truncate** sentences at the beginning or end, or choose to pad at the beginning or end as well.
- N = # samples, T = sequence length, and X[n, t] is the word / index that appears in document n, time step t.
- Remember: you first have an **NxT** matrix of word indices, that pass through an embedding layer to get an **NxTxD** tensor (where each word index is converted to a word vector), that will be subsequently passed into a NN.



Text Preprocessing

Libraries

```
%tensorflow_version 2.x
import tensorflow as tf
print(tf.__version__)

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import
pad_sequences
```

Data

```
sentences = ["I like eggs and ham.",
             "I love chocolate and bunnies.",
             "I hate onions."]
```

Tokenization and Index Mapping

```
MAX_VOCAB_SIZE = 20000 # 20K words in Oxford dict, and 3K
words most commonly used
tokenizer = Tokenizer(num_words =MAX_VOCAB_SIZE)
tokenizer.fit_on_texts(sentences)
sequences = tokenizer.texts_to_sequences(sentences)

print(sequences) # note how int start from 1, and not use
0 cos reserved for padding

# Word to Index Mapping
tokenizer.word_index
```

Processing

```
data = pad_sequences(sequences) # by default,
padding is pre, max seq length is max sentence
length
print(data)

MAX_SEQUENCE_LENGTH = 5
data = pad_sequences(sequences, maxlen =
MAX_SEQUENCE_LENGTH)
print(data)

data = pad_sequences(sequences, maxlen =
MAX_SEQUENCE_LENGTH, padding = "post")
print(data)

data = pad_sequences(sequences, maxlen = 6)
print(data)

# truncation
data = pad_sequences(sequences, maxlen = 4) #
default truncate is pre as RNN pays more attention
to final values
print(data)

# truncation
data = pad_sequences(sequences, maxlen = 4,
truncating = 'post')
print(data)
```

Spam Detection with RNN

Libraries

```
%tensorflow_version 2.x
import tensorflow as tf
print(tf.__version__)

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import
pad_sequences
from tensorflow.keras.layers import Dense, Input,
GlobalMaxPooling1D
from tensorflow.keras.layers import LSTM, Embedding
from tensorflow.keras.models import Model
```

Data

```
!wget https://lazyprogrammer.me/course_files/spam.csv

df = pd.read_csv('spam.csv', encoding = 'ISO-8859-1')
df = df.drop(['Unnamed: 2', 'Unnamed: 3', 'Unnamed: 4'],
axis = 1)
df.columns = ['labels', 'data']

# creating binary labels
df['b_labels'] = df['labels'].map({'ham': 0, 'spam': 1})
Y = df['b_labels'].values
```

Preprocessing

```
# train_test_split
df_train, df_test, Ytrain, Ytest =
train_test_split(df['data'], Y, test_size = 0.33)

# Data Preprocessing
MAX_VOCAB_SIZE = 20000
tokenizer = Tokenizer(num_words=MAX_VOCAB_SIZE)
tokenizer.fit_on_texts(df_train) # only fit on trainset
sequences_train = tokenizer.texts_to_sequences(df_train)
sequences_test = tokenizer.texts_to_sequences(df_test)

# Integer Mapping
word2idx = tokenizer.word_index # dict of word:index
V = len(word2idx)
print('Found %s unique tokens.' % V)

# Pad Sequences to get NxT matrix
data_train = pad_sequences(sequences_train)
print('Shape of data train tensor:', data_train.shape)

# Sequence length
T = data_train.shape[1]

data_test = pad_sequences(sequences_test, maxlen=T) #
possibly truncates future sentences
print('Shape of data test tensor:', data_test.shape)
```

Modelling

```
# Choose embedding dimensionality
D = 20 # think of a one-hot encoding matrix size 20
to specify the word. Hyperparameter (no right or
wrong).

# Hidden state dimensionality
M = 15

# We want the size of the embedding to (V+1) x D,
because the first index starts from 1 and not 0.
# Thus, if the final index of the embedding matrix
is V, then the array actually must have size V+1.

i = Input(shape=(T,))
x = Embedding(V+1, D)(i) # takes in sequences of
integers and returns sequences of NxTxD word
vectors
x = LSTM(M, return_sequences=True)(x)
x = GlobalMaxPooling1D()(x)
x = Dense(1, activation='sigmoid')(x)
model = Model(i, x)

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

print('Training model...')
r = model.fit(
    data_train,
    Ytrain,
    epochs = 10,
    validation_data=(data_test, Ytest)
)
```

Evaluation

```
plt.plot(r.history['loss'], label = 'loss')
plt.plot(r.history['val_loss'], label = 'val_loss')
plt.legend()

plt.plot(r.history['accuracy'], label = 'acc')
plt.plot(r.history['val_accuracy'], label =
'val_acc')
plt.legend()
```

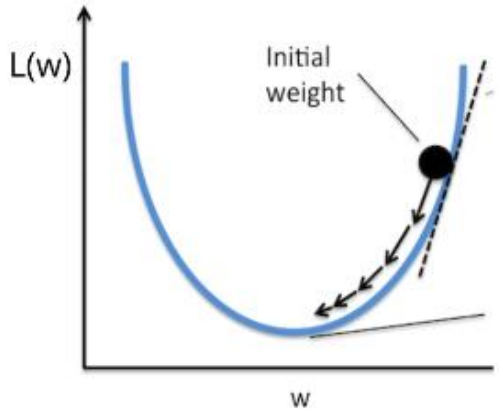
Chapter: Something

Chapter: Mean Squared Error (MSE)

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

- Mean Squared Error (MSE) is the loss function for linear regression.
    - We square the error so that it is always positive, and that there is no cancelling out effect (a positive and negative error cancelling each other out – we don't want this to happen).
    - Uses the normal Gaussian distribution.
    - Solution uses the **maximum likelihood solution**.
- Binary Cross-Entropy is the loss function for binary classification.
    - Uses the Bernoulli distribution.
    - Solution uses the **maximum likelihood solution**.
- Categorical Cross-Entropy is for multiple categorical classification.
    - Regular categorical CE uses the full NxK target, which requires NxK multiplications / additions
    - Sparse categorical CE uses the original target (1D array), which requires only N multiplications / additions

Chapter: Gradient Descent




```
w = random value
for i in range(num_epochs):
  w = w - η∇L(w)
```

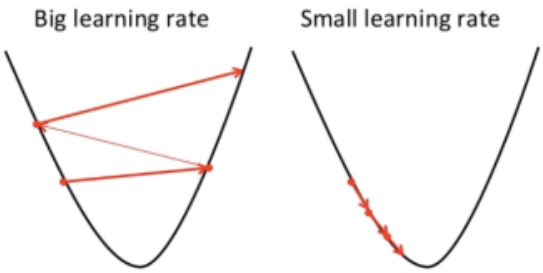
- It is the method used to train **all** models
- The purpose is to minimise the cost (L) with respect to the parameters (w).
- To minimise a function, we find its derivative and set it to zero. Where the slope is zero, that is its minimum / maximum. In ML, it is always a minimum.
- However in ML, some equations are not solvable – you cannot just take the gradient, set it to zero, then solve for w. Hence, the best way would be to approximate the answers!
- The idea is: repeatedly take small steps in the direction of the gradient to find a new w. At each step, L(w) decreases provided the step size is small enough. Eventually, it will converge to the minimum.
- The code is on the left hand side. It is really just that.

An example is:

- Minimize $L(w) = w^2$
- Gradient is dL/dw = 2w
- Inside the loop: w = w - eta * 2 * w
- The solution is w = 0, where L(w) = 0 also

Hyperparameters

- Each iteration of the loop is called an *epoch*, and we must choose the *number of epochs* high enough so the cost converges
- η (Greek letter eta) is the learning rate must be small enough so that the cost does not blow up, but large enough so that you don't have to wait longer than necessary
- Trial and error is how you choose them
- You can also use automated methods like Bayesian optimization, but that is outside the scope of this course
- *Practice* is best!



Stochastic Gradient Descent

- This just means **stochastic batch gradient descent** of small batch sizes like 32, 64, or 128.
- The idea is: we are going to split our data into **batches**, and for each batch we are going to take one step of gradient descent. If batch size is 32, then number of epochs is N / 32. On each epoch, we will make one pass through the entire dataset, with each epoch having a randomised dataset to prevent learning unwanted stuff.
- In other words, the normal "full gradient descent" has batch size of N. With batch gradient descent, the training process converges **much faster**, and for extremely huge datasets, sometimes even one step of gradient descent cannot even fit into memory.

Momentum

- This is an extremely important part in improving plain SGD, with the author saying it is an "80% factor".
- Metaphor: like pushing a box on almost-frictionless ice, it continues to move without additional force. Will slow down eventually (also occurs in ML too).
- **Gradient descent without momentum** is like trying to push a box on sand – each time you want it to move, you have to push it again. Similarly in ML, each time you want your thing to move, there has to be a gradient so we can move in the direction of the gradient.
- Can set momentum at 0.9.

Gradient Descent, without Momentum

$$\theta_t \leftarrow \theta_{t-1} - \eta g_t$$

- If g_t is 0, the parameter doesn't change!

Gradient Descent with Momentum

2 steps:

"Sliding on ice"

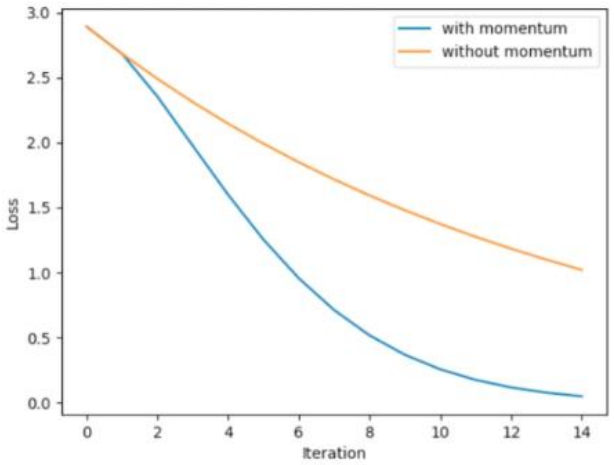
$$v_t \leftarrow \mu v_{t-1} - \eta g_t$$
$$\theta_t \leftarrow \theta_{t-1} + v_t$$

Pushing the box

Typical values of μ are 0.9, 0.95, 0.99, ...
Without any g, the box "slows down"

Basically, momentum helps to make cost converge to its minimum much faster:

Effect of Momentum



Learning Rates

Variable Learning Rates

- Learning rate is a function of time, e.g. η(t) or **learning rate scheduling**
  - Method 1: **Step Decay**, where periodically eg every 100 steps, we reduce the LR by a constant factor such as half.
  - Method 2: **Exponential Decay**, where the LR follows an exponential curve.
  - Method 3: LR Decay to 1/t: η(t) = A / (kt + 1), where the dropoff is slower than exponential decay. You can control how fast or slow the LR decays by changing the proportionality constant.
- These three methods have LR decreasing with time. Why? Initially, weights are far from optimal. But as we get closer to the minimum, learning rate may be too large and we may be oscillating / bounce back and forth / might bounce until your loss increase!

Adaptive Learning Rate Techniques

- These LR adapt to the training data, eg AdaGrad.

- **AdaGrad**: We cannot expect the dependence of cost on each parameter to be the same. In other words, in one direction the gradient might be really steep, or flat in another. It may be beneficial to adapt the LR for each parameter individually, based on how much "learning" it has done so far / based on how much it has changed in the past. But researchers have found that the LR would approach zero too quickly when there were still learning to be done. Hence, we use others.
- **RMSProp**: AdaGrad's cache is growing too fast, so we decrease it on each update to make it grow less fast.

Summary Pseudocode (for your convenience)

AdaGrad

At every batch:

```
cache += gradient ** 2
param = param - learning_rate * gradient / sqrt(cache + epsilon)
```

RMSProp

At every batch:

```
cache = decay * cache + (1 - decay) * gradient ** 2
param = param - learning_rate * gradient / sqrt(cache + epsilon)
```

epsilon = 10^{-8}, 10^{-9}, 10^{-10}, etc..., decay = 0.9, 0.99, 0.999, 0.9999, etc...

Adam

- Uses the concept of **exponentially-smoothed averages**:
  - Imagine you want to find the sample mean, so you record x_1, x_2, …, x_T.
  - But suppose you have so much data, you cannot store all the x_t at the same time.
  - You can rearrange the formula some way. It looks like the cache of RMSProp.
  - This cache of RMSProp is really estimating the average of the squared gradient. Because we eventually use the square root of the cache, now you can see where RMSProp gets its name (RMS = root mean square).
  - Variance $V = E(X^2)$. Remember this formula!
- Strangely, experimental results show that regular gradient descent perform better than all these adaptive techniques! So yes, fundamentals > adam! Strange!

Recap

- Adam is a modern adaptive learning rate technique that is the default for many deep learning practitioners
- Combines RMSProp (cache mechanism) with momentum (keeping track of old gradients)
- Generalize as 1st and 2nd moments
- Apply bias correction

$$\hat{m}_t = \frac{m_t}{1-\beta_1{}^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2{}^t} \qquad \theta_{t+1} \leftarrow \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}$$