Applied Statistical Methods

Deep Learning

Xuemao Zhang
East Stroudsburg University

April 24, 2023

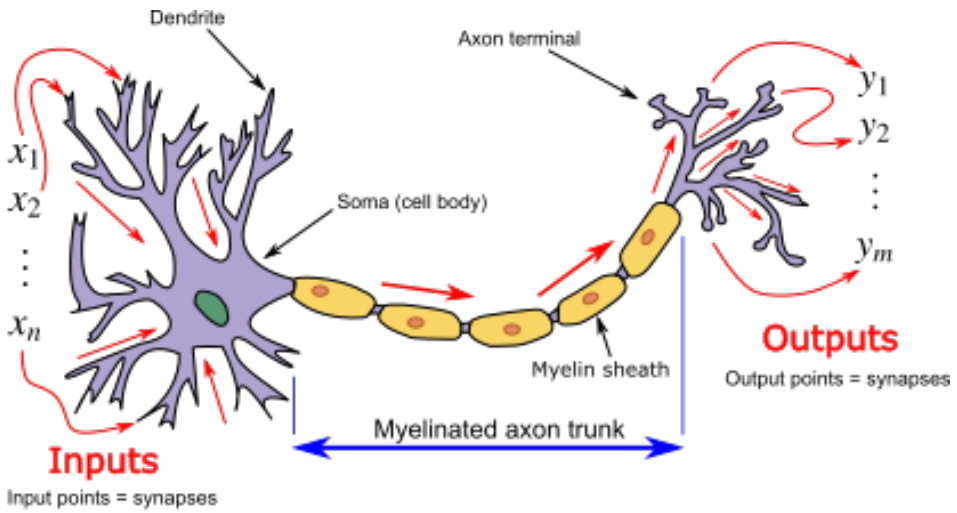
Outline

- Introduction to Neural Networks/Deep Learning
- Applications of Artificial Neural Networks
- Lab: Single Layer Network on Hitters Data
- Lab: Multilayer Network on the MNIST Digit Data
- Lab: IMDb Document Classification
- Convolution
- Pooling layers
- Convolutional Neural Networks
- Lab: CNN to the CIFAR data
- Lab: Using Pretrained CNN Models
- Recurrent Neural Networks
- RNN for time-series data
- Summary

Introduction to Neural Networks/Deep Learning

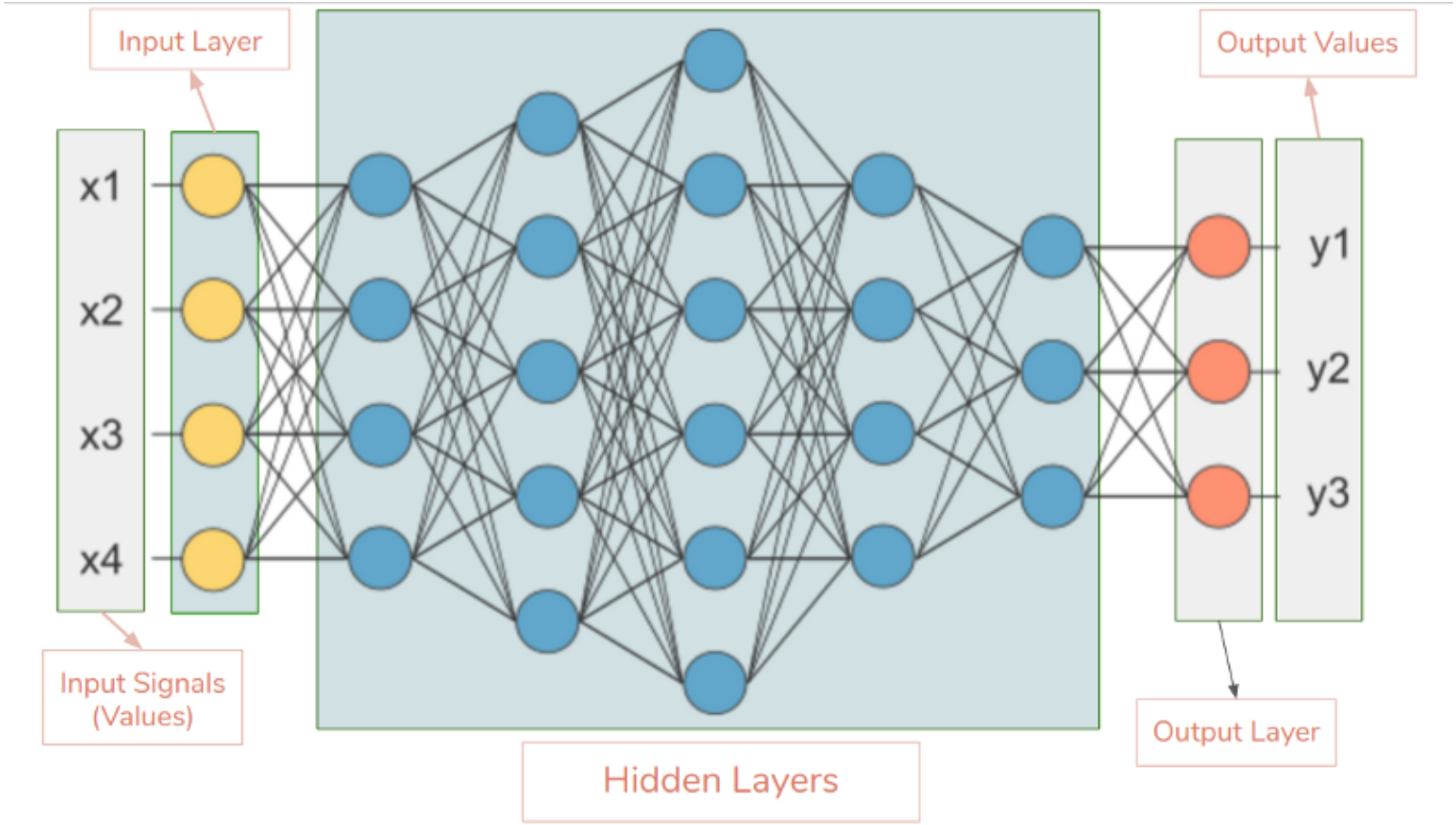
A Biological Neuron: In human brain, billions of neurons interact with each other.

- Dendrite: Receives signals from other neurons
- Soma: Processes the information
- Axon: Transmits the output of this neuron
- Synapse: output points



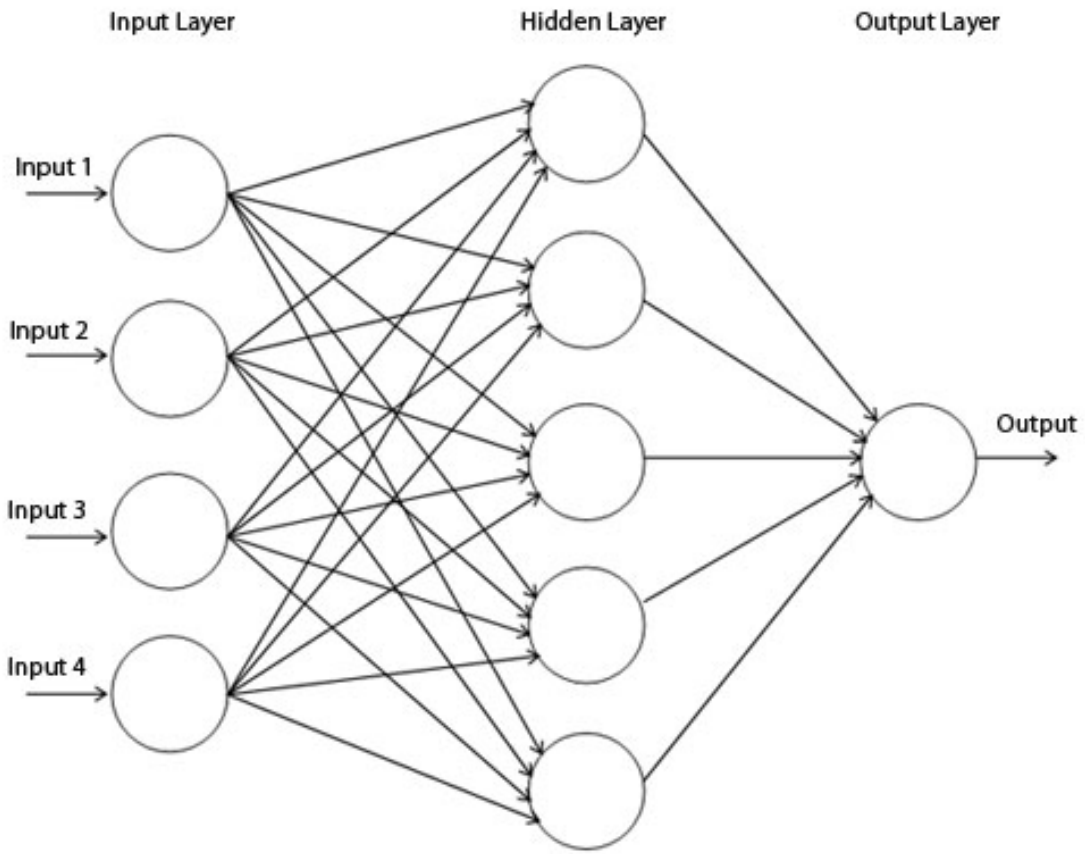
Introduction to Neural Networks/Deep Learning

- Idea is to replicate neurons in brain through Artificial Neuron.
- These artificial neurons interact with each other.

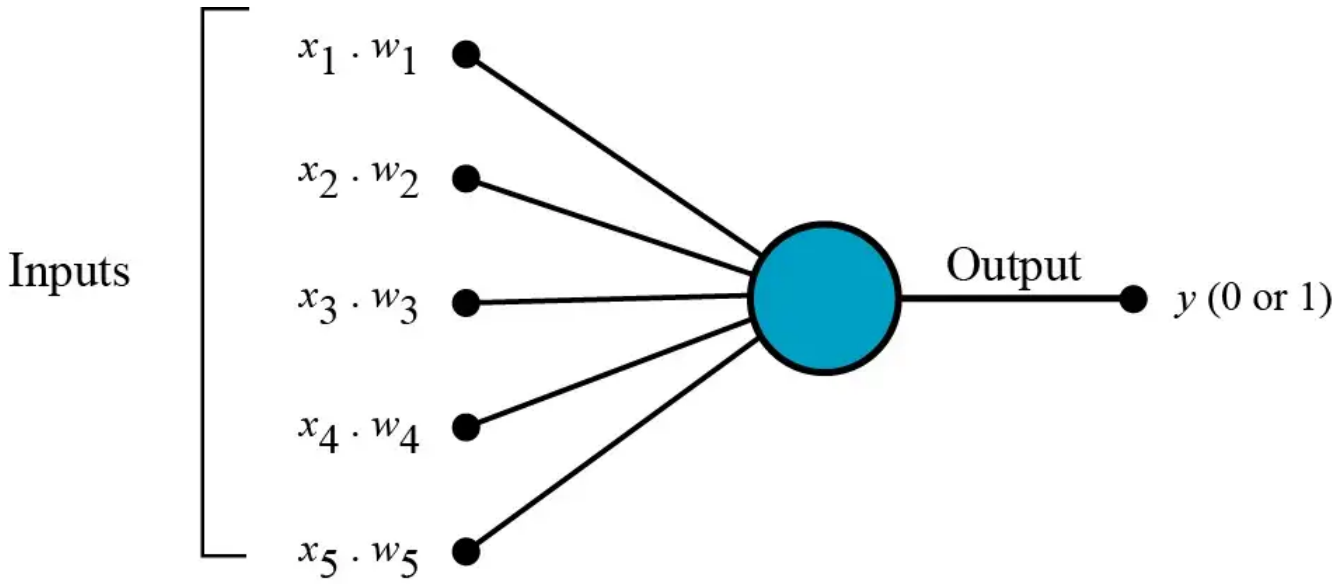


Introduction to Neural Networks/Deep Learning

- 1950's: *Perceptron*, first neuron was developed by Rosenblatt: Single layer neural network.

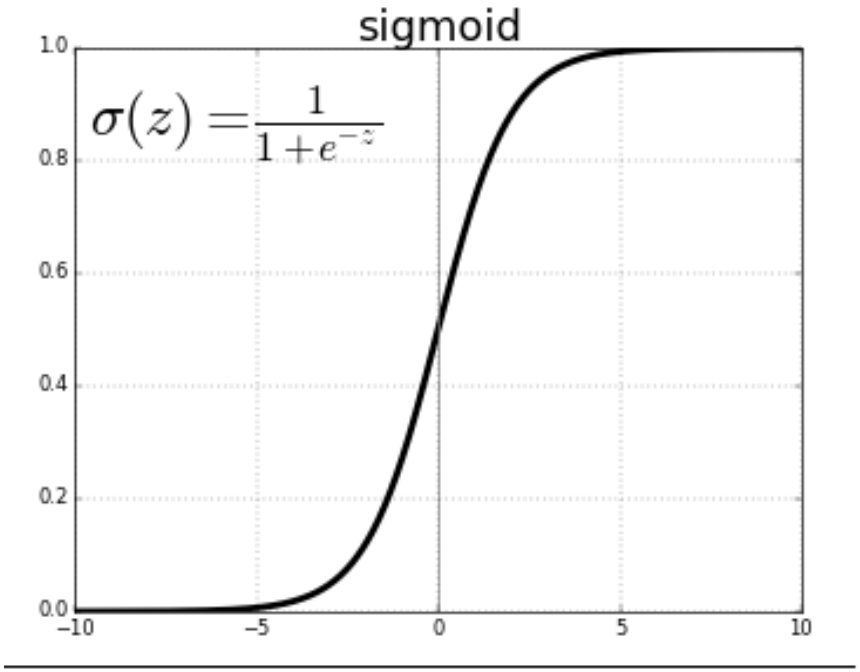


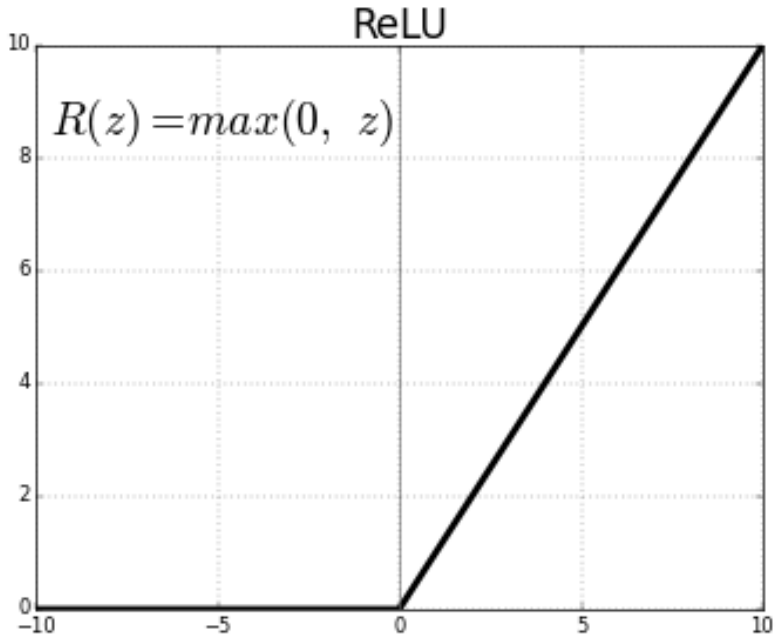
Introduction to Neural Networks/Deep Learning



Introduction to Neural Networks/Deep Learning

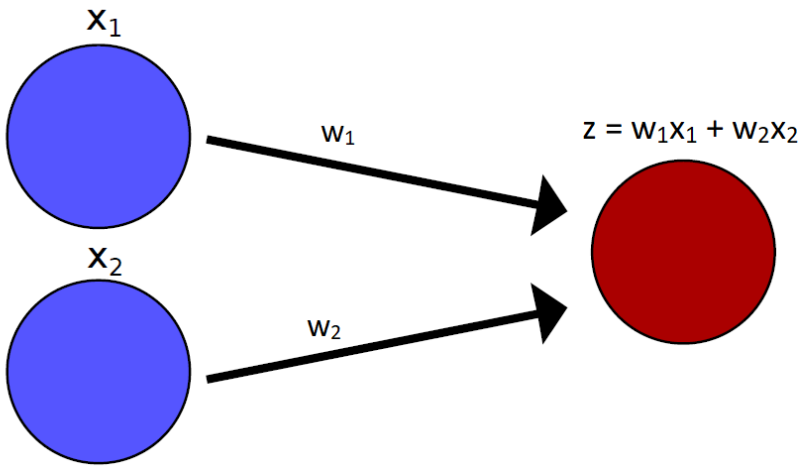
- How does a *Perceptron* work?
- Apply an **Activation Function** S to the weighted sum, and the output y is resulted from a Sigmoid function or ReLU (a step function):





Introduction to Neural Networks/Deep Learning

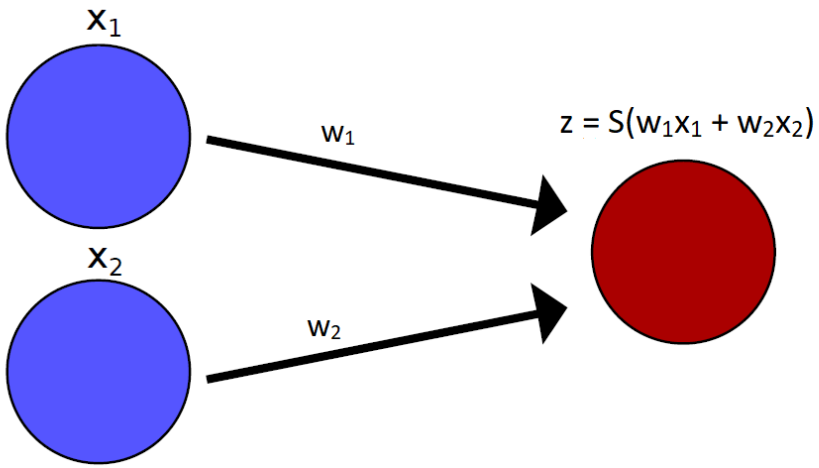
- A *Perceptron* without an **Activation Function**:



$$f_{w_1,w_2}(x_1, x_2) = w_1 x_1 + w_2 x_2$$

Introduction to Neural Networks/Deep Learning

- The **Activation Function** generally is nonlinear.



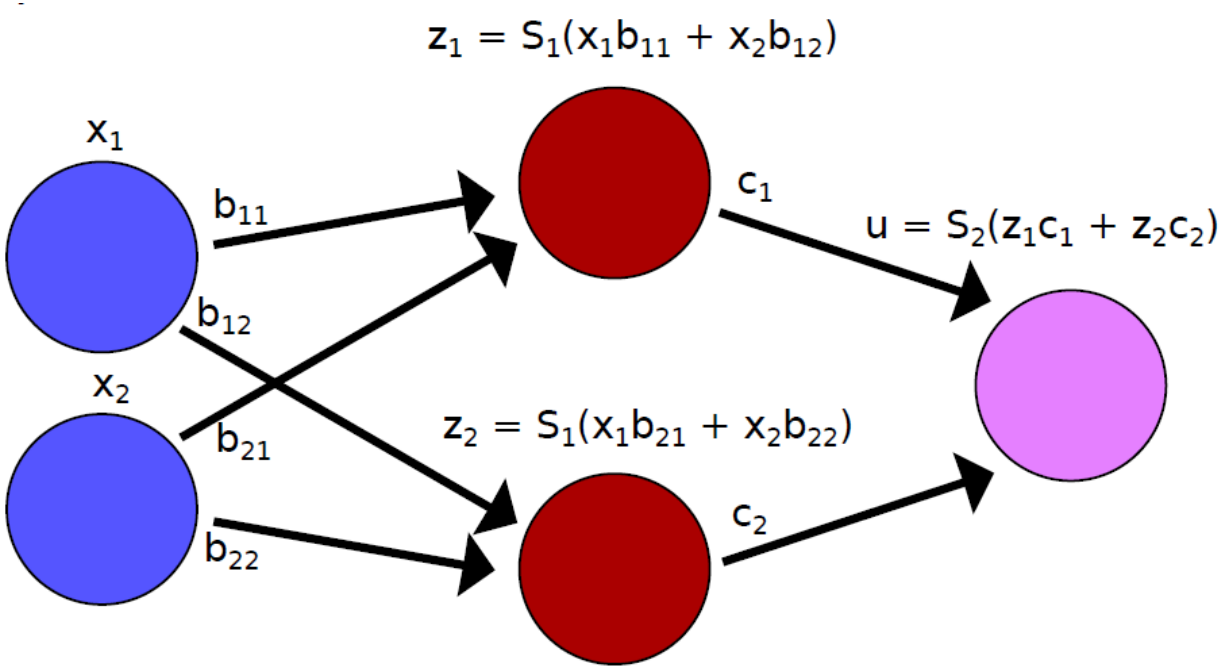
$$f_{w_1,w_2}(x_1, x_2) = S(w_1 x_1 + w_2 x_2)$$

Introduction to Neural Networks/Deep Learning

- The above simple examples connect NNs back to simple objects: linear or logistic regression without harnessing the power of neural networks.
- 1970's: The Quiet Years Limitations of Perceptron was demonstrated by Minsky and Papert (1969).
- 1980's: Renewed Interest in neural networks: Geoffrey Hinton et. al (1986) proposed a **multilayer** neural network and demonstrated the BP(backpropagation) algorithm.
  - Godfather of Deep learning.
  - At the time, he was PostDoc at UCSD.
  - Emeritus Prof. at University of Toronto; Worked at Google Brain; Chief scientific advisor of the Vector Institute in Toronto.

Introduction to Neural Networks/Deep Learning

- The power of neural networks comes through hidden layers.
- In the following, b_{11}, b_{12}, b_{21} and b_{22} are the weights for the first layer, c_1 and c_2 are the weights for the hidden layer.



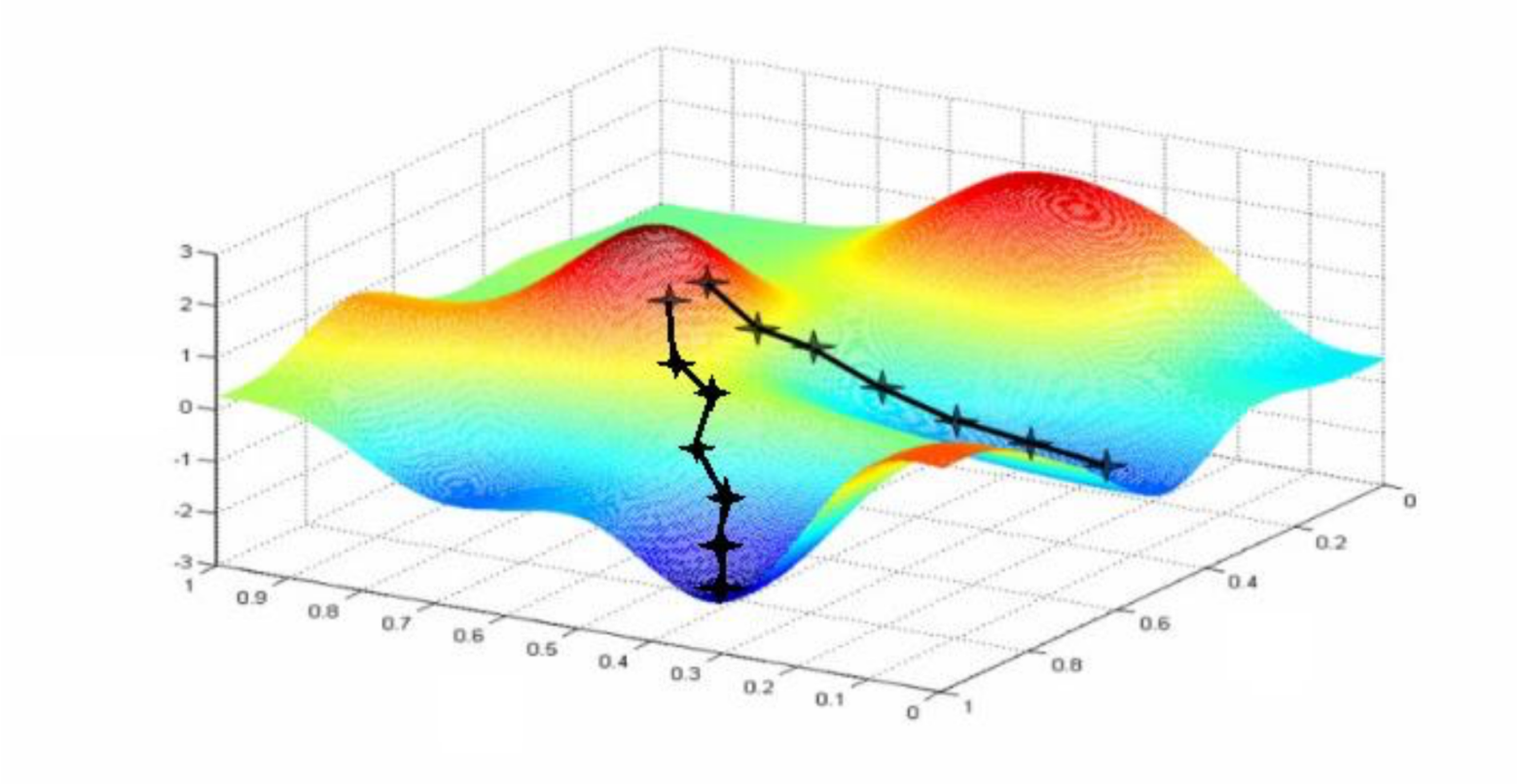
Introduction to Neural Networks/Deep Learning

- The structure of a multilayer NN:
  - Number of hidden layers
  - Number of nodes per layer
  - Types of activation functions
- For a given structure of a NN, let $\boldsymbol{\theta}$ denote the vector of coefficient values. We can think of the NN as a **function** $f_{\boldsymbol{\theta}}(\mathbf{x})$
- We choose the $\boldsymbol{\theta}$-value that best fits our data!

$$\hat{\boldsymbol{\theta}} = \arg min_{\boldsymbol{\theta}} \sum_{i=1}^{n} (y_i - f_{\boldsymbol{\theta}}(\mathbf{x_i}))^2$$

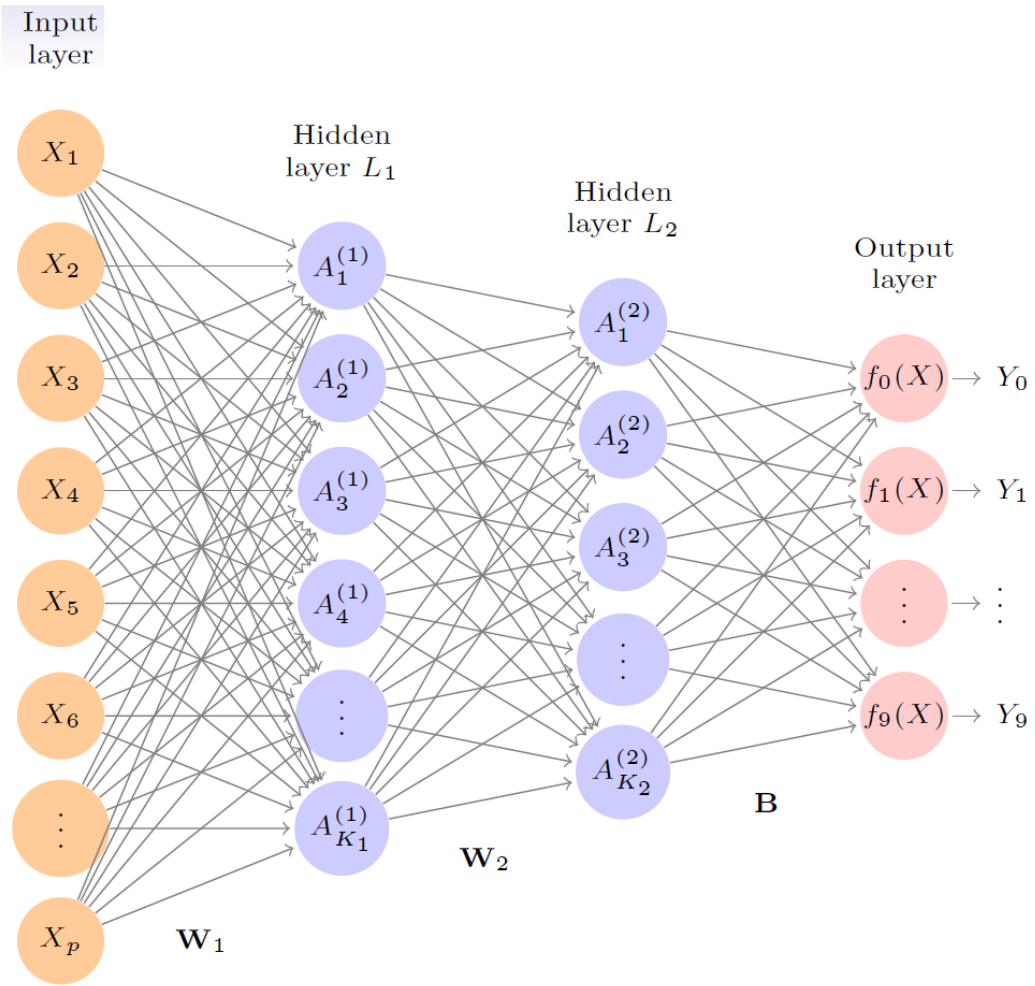
Introduction to Neural Networks/Deep Learning

- The minimizer $\hat{\theta}$ is obtained by the BP algorithm which uses gradient descent!



Introduction to Neural Networks/Deep Learning

- Example: Neural network diagram with two hidden layers and multiple outputs, suitable for the MNIST handwritten-digit problem. The input layer has $p = 784$ units, the two hidden layers $K1 = 256$ and $K2 = 128$ units respectively, and the output layer 10 units. Along with intercepts (referred to as biases in the deep-learning community) this network has 235,146 parameters (referred to as weights).



Introduction to Neural Networks/Deep Learning

- 1989: ConvNet, Yann Lecun came up with CNN (Convolutional Neural Network).
  - Prof. at New York University
  - Chief AI Scientist at Meta
  - Independently discovered BP Algorithm.
  - LeCun received the 2018 Turing Award (often referred to as "Nobel Prize of Computing"), together with Yoshua Bengio and Geoffrey Hinton, for their work on deep learning.
- Research in Neural Networks died between 1990 and 2012 because
  - Required a lot of data.
  - Computation Intensive.

Introduction to Neural Networks/Deep Learning

- It all began in 2012. The algorithm existed since 1990's, so why?
- The two problems solved
  - Lots of Data - with the help of Internet/Mobile Devices
  - Lots of computational power - GPU
- Most common variations of neural network architectures are:
  - Multilayer perceptron(MLP)
  - Convolutional Neural Network(CNN)
  - Recurrent Neural Network(RNN)

Introduction to Neural Networks/Deep Learning

- Multilayer Perceptron:
  - We build most of our fundamental understanding with Multilayer Perceptron(MLP)
- Convolutional Neural Network:
  - We will extend understanding of MLP into CNN.
  - CNN's are typically used in Images/Videos related problems.
  - Can be used to generate/draw images as well.
- Recurrent Neural Network:
  - Typically used to understand sequences, eg speech, text, etc.
  - It can even be used to generate music.

Applications of Artificial Neural Networks

- Visual Recognition
  - Recognize digits
  - Recognize letters, eg. Plate number
  - face recognition

Applications of Artificial Neural Networks

- Photo descriptions

"man in black shirt is playing guitar."

"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

Image Captioning

Applications of Artificial Neural Networks

- Speech recognition

AMAZON'S ALEXA GOOGLE'S ASSISTANT APPLE'S SIRI MICROSOFT'S CORTANA

Applications of Artificial Neural Networks

- Self-Driving Cars

Applications of Artificial Neural Networks

- Natural Language Processing
  - summarizing articles
  - machine translation
  - question answering
  - text classification
- Music composition
- Automatic Game Playing
- Applications in healthcare: predictive models, imaging techniques
- For more, see Top 26 Applications of Deep Learning in 2023

Lab: Single Layer Network on Hitters Data

- We separate the data Hitters as a training and test set.

```
import pandas as pd
import numpy as np
Hitters=pd.read_csv("../data/Hitters.csv", header=0, na_values='NA')
Hitters = Hitters.dropna()
dummies = pd.get_dummies(Hitters[['League', 'Division', 'NewLeague']])
y = Hitters['Salary']
X_ = Hitters.drop(['Salary', 'League', 'Division', 'NewLeague'],
axis=1).astype('float64')
X = pd.concat([X_, dummies[['League_N', 'Division_W', 'NewLeague_N']]], axis=1)

n=Hitters.shape[0] #sample size
p=Hitters.shape[1] #20
np.random.seed(1)
train = np.random.choice(n, int(n/2), replace=False)
select = np.in1d(range(n), train)

X_train=X[select]
y_train=y[select]
X_test=X[~select]
y_test=y[~select]
```

Lab: Single Layer Network on Hitters Data

- The linear model should be familiar, but we present it anyway.

```
import statsmodels.api as sm
lm_fit=sm.OLS(y_train, X_train).fit()
print(lm_fit.summary())
```

```
##                                OLS Regression Results
## ==============================================================================
## Dep. Variable:                 Salary   R-squared (uncentered):                   0.825
## Model:                            OLS   Adj. R-squared (uncentered):              0.795
## Method:                 Least Squares   F-statistic:                              27.80
## Date:                Tue, 11 Apr 2023   Prob (F-statistic):                    9.67e-34
## Time:                        09:45:42   Log-Likelihood:                         -926.58
## No. Observations:                 131   AIC:                                      1891.
## Df Residuals:                     112   BIC:                                      1946.
## Df Model:                          19
## Covariance Type:            nonrobust
## ==============================================================================
##                  coef    std err          t      P>|t|      [0.025      0.975]
## ------------------------------------------------------------------------------
## AtBat          -1.3256      0.855     -1.550      0.124      -3.021       0.369
## Hits            6.9383      3.359      2.066      0.041       0.284      13.593
## HmRun          10.2435      8.507      1.204      0.231      -6.613      27.100
## Runs           -2.9744      4.296     -0.692      0.490     -11.487       5.538
## RBI            -4.0546      3.651     -1.111      0.269     -11.288       3.179
## Walks           7.5861      2.737      2.772      0.007       2.163      13.009
## Years          -6.6669     14.301     -0.466      0.642     -35.002      21.669
## CAtBat          0.0821      0.195      0.422      0.674      -0.304       0.468
## CHits          -0.6762      0.877     -0.771      0.442      -2.413       1.061
## CHmRun         -0.6428      2.258     -0.285      0.776      -5.117       3.831
```

Lab: Single Layer Network on Hitters Data

- Performance on the test data

```
lpred= lm_fit.predict(X_test) #predicted values on test data
np.mean(abs(y_test-lpred)) #prediction error
```

```
## 222.0633164437715
```

- Next we fit the lasso regression

```
from sklearn.linear_model import Lasso, LassoCV
lassocv = LassoCV(alphas = None, cv = 10, max_iter = 10000)
lassocv.fit(X_train, y_train) #lassocv.alpha_
```

```
## LassoCV(cv=10, max_iter=10000)
```

```
lasso = Lasso(max_iter = 10000)
lasso.set_params(alpha=lassocv.alpha_)
```

```
## Lasso(alpha=6825.513159376968, max_iter=10000)
```

```
lasso.fit(X_train, y_train)
```

```
## Lasso(alpha=6825.513159376968, max_iter=10000)
```

```
np.mean(abs(y_test- lasso.predict(X_test)))
```

```
## 244.74772309524815
```

Lab: Single Layer Network on Hitters Data

- TensorFlow is an open-sourced end-to-end platform, a library for multiple machine learning tasks, while Keras is a high-level neural network library that runs on top of TensorFlow.
- TensorFlow is tested and supported on **Python 3.7–3.10**
  - Windows 7 or later (with C++ redistributable)
  - macOS 10.12.6 (Sierra) or later (no GPU support)
- If you have Python 3.11 installed on your computer, the simplest solution is to uninstall it first.
- We install Python 3.10.9 https://www.python.org/downloads/release/python-3109/
  - Choose *Windows installer (64-bit)* for Windows
  - Choose *macOS 64-bit universal2 installer* for mac
- In Windows *Command Prompt*:

```
pip install --upgrade pip
pip install tensorflow
```

Lab: Single Layer Network on Hitters Data

- In the table of statistics it's easy to see how different the ranges of each feature are:

```
X_train.shape
```

```
## (131, 19)
```

```
(X_train.iloc[:, :16]).describe().transpose()[['mean', 'std']]
```

```
##                mean          std
## AtBat     415.519084   147.737676
## Hits      110.473282    43.725952
## HmRun      11.816794     9.492346
## Runs       56.190840    25.013878
## RBI        51.717557    25.982916
## Walks      42.824427    20.308953
## Years       7.244275     4.824762
## CAtBat   2698.748092  2265.419144
## CHits     729.709924   641.795864
## CHmRun     66.496183    75.412749
## CRuns     370.687023   336.960876
## CRBI      320.541985   291.544990
## CWalks    272.381679   279.558951
## PutOuts   291.351145   257.129208
## Assists   132.748092   147.592701
## Errors      9.419847     6.757280
```

Lab: Single Layer Network on Hitters Data

- Normalization: It is **good practice to normalize continuous features** that use different scales and ranges.
  - One reason this is important is because the features are multiplied by the model weights. So, the scale of the outputs and the scale of the gradients are affected by the scale of the inputs.
- The Normalization layer: The `tf.keras.layers.Normalization` is a clean and simple way to add feature normalization into your model.
  - https://www.tensorflow.org/api_docs/python/tf/keras/layers/Normalization
- It is not meaningful to normalize dummy variables. But it does not affect predictions.

Lab: Single Layer Network on Hitters Data

- You may manually normalize your data. And use the normalized data for training and test as well

```
X_train1=X_train.iloc[:, :16]
mean_train1=X_train1.mean(axis=0)
std_train1=X_train1.std(axis=0)
X_train1 -= mean_train1
X_train1 /= std_train1
#Then merge with the dummy variables by joining columns
X_train_normal = X_train1.join(X_train.iloc[:, 16:])

X_test1=X_test.iloc[:, :16]
mean_test1=X_test1.mean(axis=0)
std_test1=X_test1.std(axis=0)
X_test1 -= mean_test1
X_test1 /= std_test1
X_test_normal = X_test1.join(X_test.iloc[:, 16:])
```

Lab: Single Layer Network on Hitters Data

- import packages

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

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

```
## 2.11.0
```

```
from keras import models
from keras import layers
```

Lab: Single Layer Network on Hitters Data

- The first step is to create the **normalization layer** using `tf.keras.layers.Normalization`;
  - https://keras.io/api/layers/preprocessing_layers/numerical/normalization/
- Then, fit the state of the preprocessing layer to the data by calling Normalization.adapt:
  - During `adapt()`, the layer will compute a mean and variance separately for each position in each axis

```
import numpy as np
normalizer = tf.keras.layers.Normalization(axis=-1)
```

```
normalizer.adapt(np.array(X_train))
print(normalizer.mean.numpy())
```

```
## [[4.1551910e+02 1.1047328e+02 1.1816794e+01 5.6190838e+01 5.1717556e+01
##   4.2824429e+01 7.2442746e+00 2.6987480e+03 7.2970996e+02 6.6496185e+01
##   3.7068704e+02 3.2054199e+02 2.7238168e+02 2.9135114e+02 1.3274809e+02
##   9.4198475e+00 4.1984734e-01 5.1145041e-01 4.2748091e-01]]
```

```
np.array(X_train[:1])
```

```
## array([[3.150e+02, 8.100e+01, 7.000e+00, 2.400e+01, 3.800e+01, 3.900e+01,
##         1.400e+01, 3.449e+03, 8.350e+02, 6.900e+01, 3.210e+02, 4.140e+02,
##         3.750e+02, 6.320e+02, 4.300e+01, 1.000e+01, 1.000e+00, 1.000e+00,
##         1.000e+00]])
```

Lab: Single Layer Network on Hitters Data

- When the layer is called, it returns the input data, with each feature independently normalized:

```
first = np.array(X_train[:1])
print(first)
```

```
## [[3.150e+02 8.100e+01 7.000e+00 2.400e+01 3.800e+01 3.900e+01 1.400e+01
##   3.449e+03 8.350e+02 6.900e+01 3.210e+02 4.140e+02 3.750e+02 6.320e+02
##   4.300e+01 1.000e+01 1.000e+00 1.000e+00 1.000e+00]]
```

```
print(normalizer(first))
# dummy variables are normalized too
```

```
## tf.Tensor(
## [[-0.68300104 -0.67663306 -0.5093878  -1.2918594  -0.5299719  -0.18903534
##    1.4055943   0.3324471   0.16468512  0.03332893 -0.14802247  0.32179177
##    0.36848134  1.3299017  -0.6104138   0.08618551  1.1755077   0.9773555
##    1.1572751 ]], shape=(1, 19), dtype=float32)
```

Lab: Single Layer Network on Hitters Data

- We will build a regression model using deep learning in `Keras`. To begin with, we will define the model using a `tf.keras.Sequential` model, which represents a sequence of steps.
  - https://www.tensorflow.org/tutorials/keras/regression
  - https://www.tensorflow.org/guide/keras/sequential_model

- Normalize the training features

```
X_train_normalizer = layers.Normalization(axis=-1)
```

- `adapt()` should be called before model construction if we put a normalization layer.

```
X_train_normalizer.adapt(X_train)
```

Lab: Single Layer Network on Hitters Data

- We are using the Sequential model because our network consists of a **linear stack** of layers
- The second line of code represents the first layer which specifies the activation function and the number of input dimensions, which in our case is 19 predictors.

```
Hitters_model=tf.keras.Sequential(layers=X_train_normalizer)
#Hitters_model = keras.Sequential() #Sequential constructor without normalizing data
Hitters_model.add( layers.Dense(50, input_shape=(X_train.shape[1], ),
activation= "relu") ) #hidden layer
```

Lab: Single Layer Network on Hitters Data

- Adding dropout: *Dropout* is one of the most effective and most commonly used regularization techniques for neural networks to help reduce overfitting, developed by Geoff Hinton and his students at the University of Toronto.
- Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of *output features* of the layer during training.
- The dropout rate is the fraction of the features that are zeroed out; it's usually set *between 0.2 and 0.5*.
- At test time, no units are dropped out; instead, the layer's output values are scaled down by a factor equal to the dropout rate, to balance for the fact that more units are active than at training time.

```
Hitters_model.add(layers.Dropout(0.2)) #dropout layer
Hitters_model.add(layers.Dense(1)) #output layer
```

Lab: Single Layer Network on Hitters Data

- Model summary

```
Hitters_model.summary()
```

```
## Model: "sequential"
## _________________________________________________________________
##  Layer (type)                Output Shape              Param #
## =================================================================
##  normalization_1 (Normalizat  (None, 19)               39
##  ion)
##
##  dense (Dense)               (None, 50)                1000
##
##  dropout (Dropout)           (None, 50)                0
##
##  dense_1 (Dense)             (None, 1)                 51
##
## =================================================================
## Total params: 1,090
## Trainable params: 1,051
## Non-trainable params: 39
## _________________________________________________________________
```

Lab: Single Layer Network on Hitters Data

- Once the model is built, configure the training procedure using the Keras Model.compile method. The most important arguments to compile are the loss and the optimizer, since these define what will be optimized (mean_absolute_error) and how (using the tf.keras.optimizers.Adam).
  - Model.compile method: https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile

```
Hitters_model.compile(optimizer='rmsprop',
loss='mse', metrics='mean_absolute_error')
```

- Loss function: This should match the type of problem you're trying to solve. For example,
  - You may use `mse` for regression
  - Use `binary_crossentropy` for binary response
  - Use `categorical_crossentropy` for categorical response
- Optimization configuration: In most cases, it's safe to go with `rmsprop` and its `default learning rate`.

Lab: Single Layer Network on Hitters Data

- The final step is to supply training data, and fit the model. Use Keras model.fit to execute the training for 1000 epochs
  - https://keras.io/api/models/model_training_apis/

```
history=Hitters_model.fit(X_train, y_train,
epochs=100, #number of iterations
batch_size=32, #stochastic gradient descent size
verbose=0, #train the data in silent mode
#validation_split = 0.2,
# Calculate validation results on 20% of the training data.
validation_data=(X_test, y_test)
#validation_data will override validation_split
)
```

- The loss and any model metrics are evaluated on validation_data at the end of each epoch.

Lab: Single Layer Network on Hitters Data

- Check the fitting procedure

```
hist = pd.DataFrame(history.history)
hist.columns
```

```
## Index(['loss', 'mean_absolute_error', 'val_loss', 'val_mean_absolute_err
```

```
hist['epoch'] = history.epoch
hist.tail()
```

```
##             loss  mean_absolute_error  ...  val_mean_absolute_error  epo
## 95  418447.56250           508.128265  ...               499.038239
## 96  417195.31250           507.624390  ...               498.473083
## 97  417303.34375           507.499695  ...               497.740356
## 98  415917.46875           506.444489  ...               497.216492
## 99  414909.81250           506.370453  ...               496.462158
##
## [5 rows x 5 columns]
```

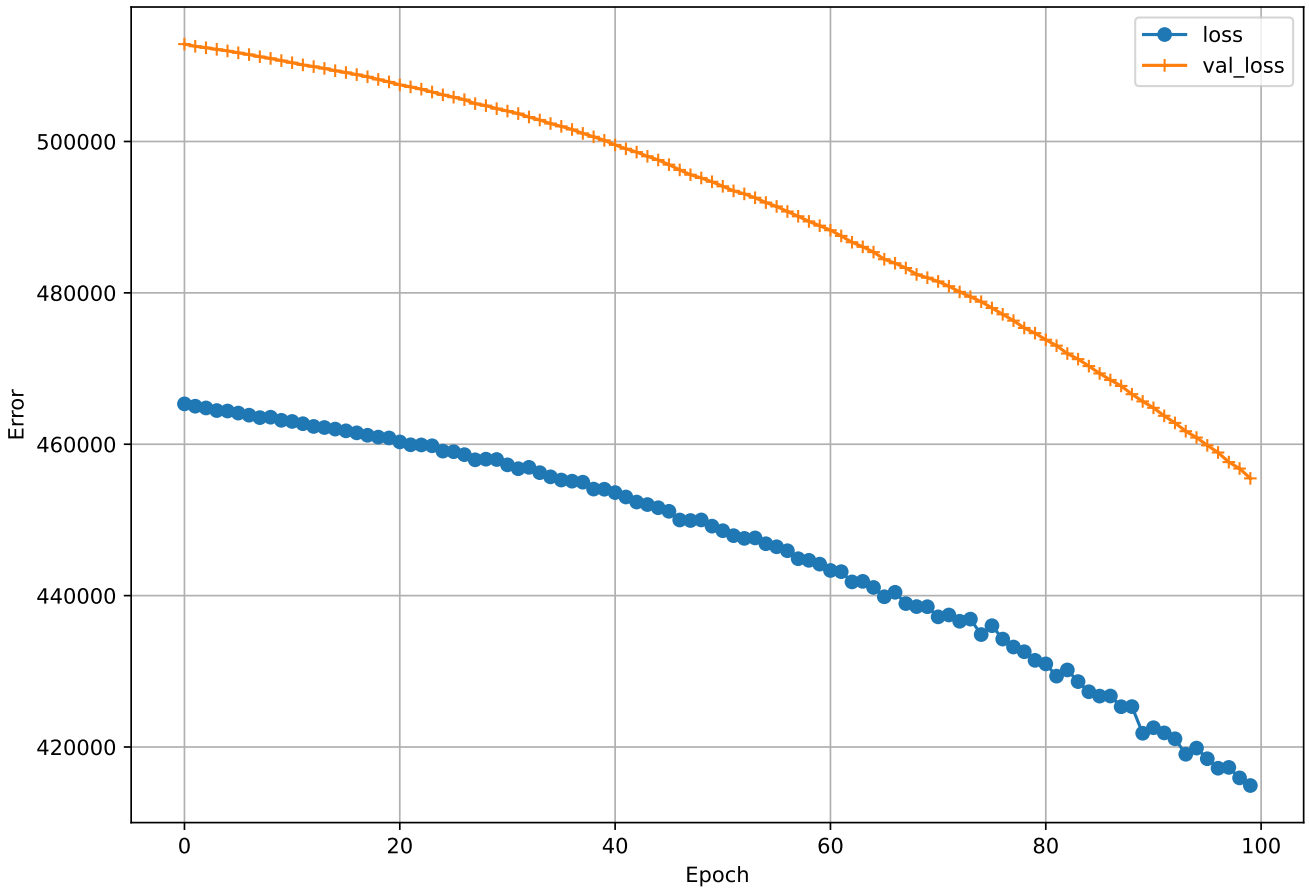
Lab: Single Layer Network on Hitters Data

- Visualize the model's training progress using the stats stored in the history object:

```
import matplotlib.pyplot as plt

def plot_loss(history):
  plt.plot(history.epoch, history.history['loss'], marker = 'o', label='loss')
  plt.plot(history.epoch, history.history['val_loss'],
  marker = '+', label='val_loss')
  plt.xlabel('Epoch')
  plt.ylabel('Error')
  plt.legend()
  plt.grid(True)
```

Lab: Single Layer Network on Hitters Data

```
plot_loss(history)
plt.show()
```



Lab: Single Layer Network on Hitters Data

- Performance on the test data

```
npred= Hitters_model.predict(X_test) #predicted values
```

```
##
## 1/5 [=====>........................] - ETA: 0s
## 5/5 [==============================] - 0s 4ms/step
```

```
np.mean(abs(y_test- npred[:,0]))
```

```
## 496.4621921702298
```

- the loss and accuracy of the network over the test data

```
test_loss, test_acc = Hitters_model.evaluate(X_test, y_test)
```

```
##
## 1/5 [=====>........................] - ETA: 0s - loss: 362782.6250 - mean_absolut
## 5/5 [==============================] - 0s 0s/step - loss: 455489.9375 - mean_abso
```

```
print('test_loss:', test_loss)
```

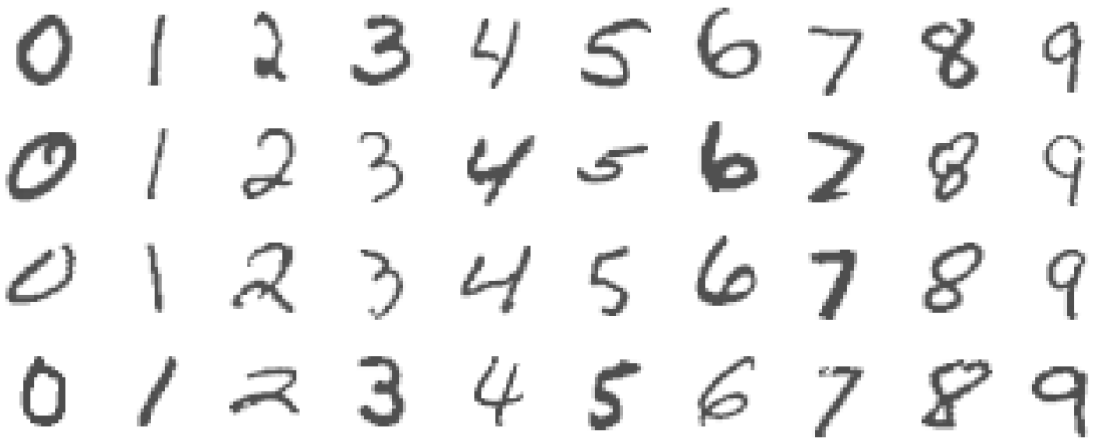
```
## test_loss: 455489.9375
```

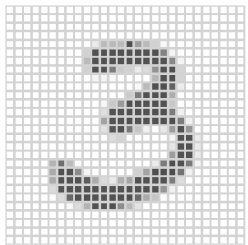
```
print('test_acc:', test_acc)
```

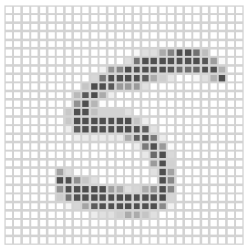
```
## test_acc: 496.462158203125
```

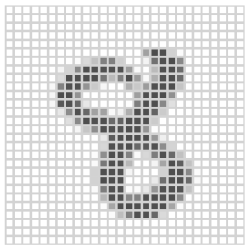
Lab: Multilayer Network on the MNIST Digit Data

- The problem to solve is to classify grayscale images of handwritten digits into their 10 categories (0 through 9).
- MNIST data
  - assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s.
  - Each grayscale image has 28×28 pixels, each of which is an eight-bit number (0–255) which represents how dark that pixel is.
  - 60,000 training, 10,000 test images



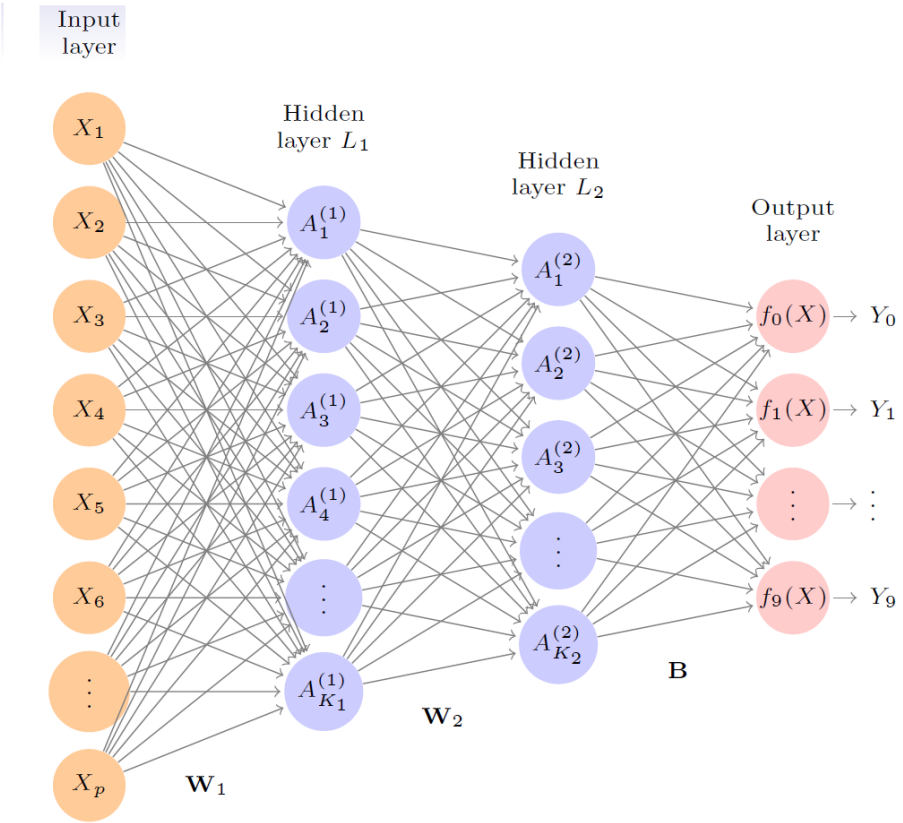






Lab: Multilayer Network on the MNIST Digit Data

- Two-layer network with 256 units at first layer, 128 units at second layer, and 10 units at output layer.
- Along with intercepts (called biases) there are 235,146 parameters (referred to as weights).



Lab: Multilayer Network on the MNIST Digit Data

- The MNIST dataset comes preloaded in Keras, in the form of a set of four Numpy arrays.
- The training images were stored in an array of shape (60000, 28, 28) of type *uint8* with values in the [0, 255] interval

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
print(len(train_images))
```

```
## 60000
```

```
print(train_images.shape)
```

```
## (60000, 28, 28)
```

```
print(train_images.dtype) #data type

#the 1st training image:
```

```
## uint8
```

```
print((train_images[0,:,:]).shape)
```

```
## (28, 28)
```

```
print(train_images[0,:,:])
```

```
## [[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##     0   0   0   0   0   0   0   0   0   0]
```

Lab: Multilayer Network on the MNIST Digit Data

- The network architecture:
  - The first layer goes from $28 \times 28 = 784$ input units to a hidden layer of 256 units, which uses the ReLU activation function.
  - The second hidden layer comes next, with 128 hidden units, followed by a dropout layer.
  - The final layer is the output layer, with activation `softmax` for the 10-class classification problem, which defines the map from the second hidden layer to class probabilities by the `softmax` activation function

$$f_m(X) = P_r(Y = m|X) = \frac{e^{Z_m}}{\sum_{j=0}^{9} e^{Z_j}}, m = 0, 1, 2, \ldots, 9.$$

```
from keras import models
from keras import layers
network = models.Sequential()
network.add(layers.Dense(256, activation='relu', input_shape=(28*28,)))
network.add(layers.Dropout(0.4)) #dropout layer
network.add(layers.Dense(128, activation='relu', input_shape=(28*28,)))
network.add(layers.Dropout(0.3)) #dropout layer
network.add(layers.Dense(10, activation='softmax'))
```

Lab: Multilayer Network on the MNIST Digit Data

- The network architecture:
  - Finally we check the model summary
  - The parameters for each layer include a bias term, which results in a parameter count of 235,146. For example, the first hidden layer involves $(784 + 1) \times 256 = 200,960$ parameters.

```
print(network.summary())
```

```
## Model: "sequential_1"
## _________________________________________________________________
##  Layer (type)                 Output Shape              Param #
## =================================================================
##  dense_2 (Dense)              (None, 256)               200960
##
##  dropout_1 (Dropout)          (None, 256)               0
##
##  dense_3 (Dense)              (None, 128)               32896
##
##  dropout_2 (Dropout)          (None, 128)               0
##
##  dense_4 (Dense)              (None, 10)                1290
##
## =================================================================
## Total params: 235,146
## Trainable params: 235,146
```

Lab: Multilayer Network on the MNIST Digit Data

- The compilation step
  - We fit the model by minimizing the cross-entropy function, the negative multinomial log-likelihood given by (10.14) in the text.

```
network.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])
```

Lab: Multilayer Network on the MNIST Digit Data

- Before we fit the model, we pre-process the data because Neural networks are somewhat sensitive to the scale of the inputs.
  - Here the inputs are eight-bit (Note: Eight bits means 2^8, which equals 256. Since the convention is to start at 0, the possible values range from 0 to 255.)
  - Grayscale values between 0 and 255, so we rescale to the unit interval: we transform the training images into a *float32* array of shape (60000, 28 * 28) with values between 0 and 1.

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32')/255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32')/255
```

- We also need to categorically encode the labels

```
from keras.utils import to_categorical
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

Lab: Multilayer Network on the MNIST Digit Data

- We're now ready to train the network by calling the `fit` method using `epochs=15`

```
history=network.fit(train_images, train_labels,
epochs=5,
batch_size=128,
verbose=0, #validation_split = 0.2
validation_data=(test_images, test_labels))
```

Lab: Multilayer Network on the MNIST Digit Data

- Two quantities are displayed during training: the loss of the network over the training data, and the accuracy of the network over the training data.
- Now let's check if the model performs well on the test set.

```
test_loss, test_acc = network.evaluate(test_images, test_labels)
```

```
##
##   1/313 [..............................] - ETA: 14s - loss: 0.0197 - acc
##  17/313 [>.............................] - ETA: 1s - loss: 0.0787 - accu
##  38/313 [==>...........................] - ETA: 0s - loss: 0.0969 - accu
##  60/313 [====>.........................] - ETA: 0s - loss: 0.1092 - accu
##  80/313 [======>.......................] - ETA: 0s - loss: 0.1108 - accu
## 102/313 [========>.....................] - ETA: 0s - loss: 0.1068 - accu
## 124/313 [==========>...................] - ETA: 0s - loss: 0.1121 - accu
## 144/313 [============>.................] - ETA: 0s - loss: 0.1119 - accu
## 166/313 [==============>...............] - ETA: 0s - loss: 0.1049 - accu
## 186/313 [================>.............] - ETA: 0s - loss: 0.1015 - accu
## 207/313 [==================>...........] - ETA: 0s - loss: 0.0989 - accu
## 224/313 [====================>.........] - ETA: 0s - loss: 0.0934 - accu
## 245/313 [======================>.......] - ETA: 0s - loss: 0.0866 - accu
## 267/313 [========================>.....] - ETA: 0s - loss: 0.0834 - accu
## 287/313 [==========================>...] - ETA: 0s - loss: 0.0802 - accu
```

Lab: Multilayer Network on the MNIST Digit Data

```
print('test_loss:', test_loss)
```

```
## test_loss: 0.0803674682974815 4
```

```
print('test_acc:', test_acc)
```

```
## test_acc: 0.9765999913215637
```

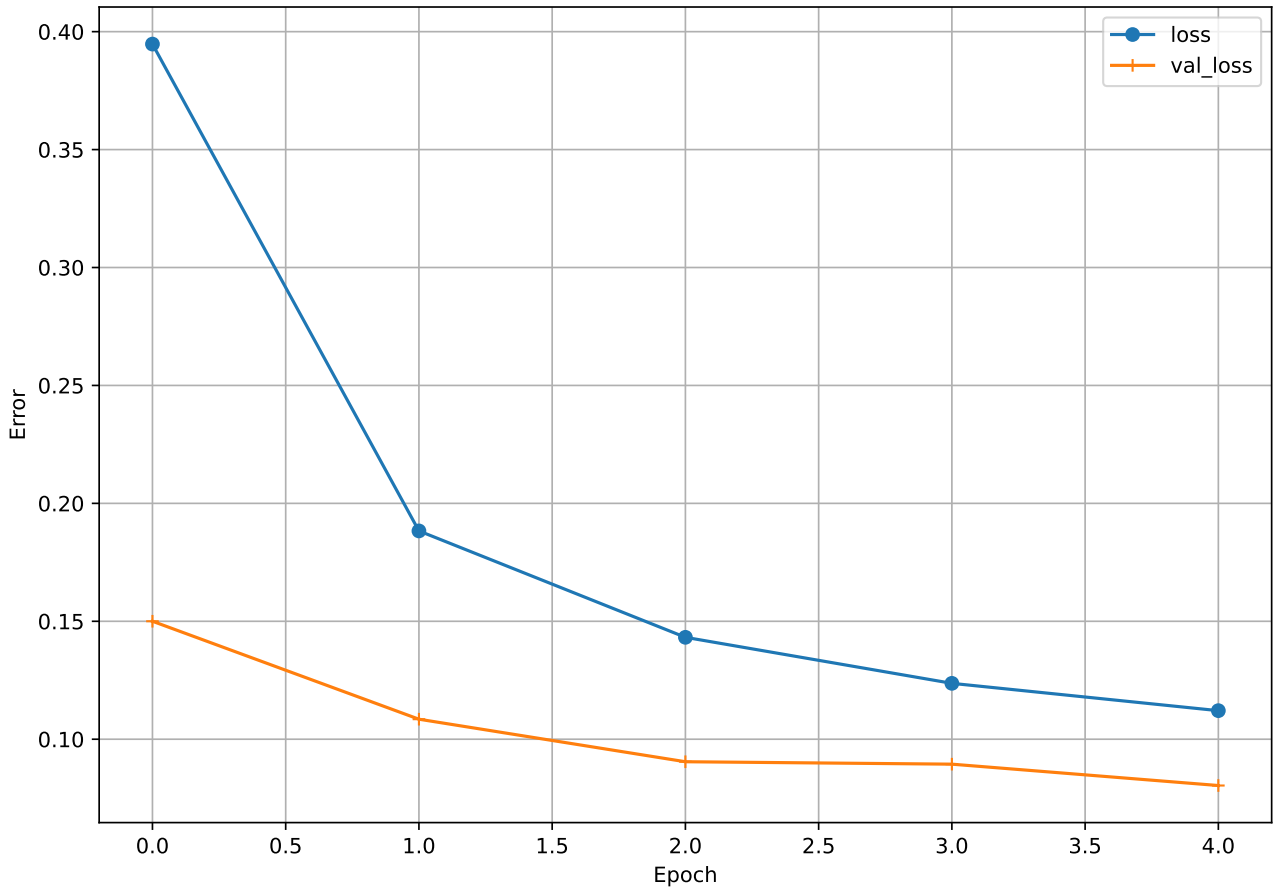
Lab: Multilayer Network on the MNIST Digit Data

- We can visualize the fitting procedure

```
import matplotlib.pyplot as plt
def plot_loss(history):
  plt.plot(history.epoch, history.history['loss'], marker = 'o',
  label='loss')
  plt.plot(history.epoch, history.history['val_loss'], marker = '+',
  label='val_loss')
  plt.xlabel('Epoch')
  plt.ylabel('Error')
  plt.legend()
  plt.grid(True)
```

Lab: Multilayer Network on the MNIST Digit Data

```
plot_loss(history)
plt.show()
```



Lab: Multilayer Network on the MNIST Digit Data

- We can use keras to fit an LDA model with much faster speed.

```
from keras import models
from keras import layers
modellr = models.Sequential()
modellr.add(layers.Dense(10, activation='softmax', input_shape=(28*28,)))
print(modellr.summary())
```

```
## Model: "sequential_2"
## _________________________________________________________________
##  Layer (type)                 Output Shape              Param #
## =================================================================
##  dense_5 (Dense)              (None, 10)                7850
##
## =================================================================
## Total params: 7,850
## Trainable params: 7,850
## Non-trainable params: 0
## _________________________________________________________________
## None
```

Lab: Multilayer Network on the MNIST Digit Data

- We fit the model just as before.

```
modellr.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])

modellr.fit(train_images, train_labels,
epochs=5, batch_size=128,
verbose=0, #validation_split = 0.2
validation_data=(test_images, test_labels))
```

```
## <keras.callbacks.History object at 0x000001F45B471930>
```

```
test_loss, test_acc = modellr.evaluate(test_images, test_labels)
```

```
##
##   1/313 [..............................] - ETA: 14s - loss: 0.2526 - acc
##  30/313 [=>............................] - ETA: 0s - loss: 0.2824 - accu
##  59/313 [====>.........................] - ETA: 0s - loss: 0.3393 - accu
##  89/313 [=======>......................] - ETA: 0s - loss: 0.3485 - accu
## 117/313 [==========>...................] - ETA: 0s - loss: 0.3348 - accu
## 142/313 [============>.................] - ETA: 0s - loss: 0.3506 - accu
## 171/313 [===============>..............] - ETA: 0s - loss: 0.3329 - accu
```

Lab: Multilayer Network on the MNIST Digit Data

```
print('test_loss:', test_loss)
```

```
## test_loss: 0.27536439895629883
```

```
print('test_acc:', test_acc)
```

```
## test_acc: 0.9244999885559082
```

Lab: IMDb Document Classification

- The `IMDB` corpus consists of user-supplied movie ratings for a large collection of movies. Each has been labeled for `sentiment` as `positive` or `negative`. Here is the beginning of a negative review:

```
This has to be one of the worst films of the 1990s. When my friends & I
were watching this film (being the target audience it was aimed at) we
just sat & watched the first half an hour with our jaws touching the
floor at how bad it really was. The rest of the time, everyone else in
the theater just started talking to each other, leaving or generally
crying into their popcorn ...
```

- We have labeled training and test sets, each consisting of 25,000 reviews, and each balanced with regard to sentiment.
- We wish to build a classifier to predict the sentiment of a review.

Lab: IMDb Document Classification

- Documents have different lengths, and consist of sequences of words. How do we create features X to characterize a document?
  - From a dictionary, identify the 10,000 most frequently occurring words.
  - Create a binary vector of length $p = 10,000$ for each document, and score a 1 in every position that the corresponding word occurred.
  - With n documents, we now have a $n \times p$ **sparse** feature matrix $\boldsymbol{X}$.
  - Bag-of-words are called `unigrams`. We can instead use `bigrams` (occurrences of adjacent word pairs, and the number of parameters is increased to p^2), and in general m-grams.

Lab: IMDb Document Classification

- Now we perform document classification on the `IMDB` dataset, which is available as part of the `keras` package. We limit the dictionary size to the 10,000 most frequently-used words and tokens.

```
from keras.datasets import imdb
from keras import preprocessing
max_features =10000 #Number of words to consider as features
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
len(x_train)
```

```
## 25000
```

```
len(x_test)
#Loads the data as lists of integers
```

```
## 25000
```

- Each element of `x_train` is a vector of numbers between 0 and 9999 (the document), referring to the words found in the dictionary. For example, the first training document is the positive review on page 419. The indices of the first 12 words are given below.

```
x_train[0][0:12]
```

```
## [1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468]
```

Lab: IMDb Document Classification

- Here's how we can quickly decode one of these reviews back to English words:
- In Python Dictionary, `items()` method is used to return the list with all dictionary keys with values https://docs.python.org/3/library/stdtypes.html?highlight=items#dict.items

```
word_index = imdb.get_word_index()
#word_index is a dictionary mapping words to an integer index.
print(type(word_index)) #dictionary
```

```
## <class 'dict'>
```

```
print(len(word_index))
#Reverses it, mapping integer indices to words
```

```
## 88584
```

```
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])

print(type(reverse_word_index)) #dictionary
```

```
## <class 'dict'>
```

```
print(len(reverse_word_index))
```

```
## 88584
```

Lab: IMDb Document Classification

- The `join()` method takes all items in an iterable and joins them into one string https://docs.python.org/3/library/stdtypes.html?highlight=join#str.join
- `get()` method returns the value of the item with the specified key https://docs.python.org/3/library/stdtypes.html?highlight=get#dict.get

```
decoded_review = ' '.join(
[reverse_word_index.get(i - 3, '?') for i in x_train[0]])
print(decoded_review)
```

```
## ? this film was just brilliant casting location scenery story direction everyone
```

- Note that the indices are offset by 3 because 0, 1, and 2 are reserved indices for `padding`, `start of sequence`, and `unknown`.

Lab: IMDb Document Classification

- One-hot encoding each document in a list of documents, and return a binary matrix in sparse-matrix format.
  - For example, turning the sequence `[3, 5]` into a 10,000-dimensional vector that would be all 0s except for indices 3 and 5, which would be 1s.

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
  results = np.zeros((len(sequences), dimension)) #all zero matrix
  for i, sequence in enumerate(sequences):
    results[i, sequence] = 1
#Sets specific indices of results[i] to 1s
  return results
```

Lab: IMDb Document Classification

- One-hot encoding each document in a list of documents, and return a binary matrix in sparse-matrix format.

```
x_train_1h = vectorize_sequences(x_train)
x_test_1h = vectorize_sequences(x_test)
print(x_train_1h.shape)
```

```
## (25000, 10000)
```

```
x_train_1h.sum()/(25000*10000)
```

```
## 0.013169872
```

- Only 1.3% of the entries are nonzero, so this amounts to considerable savings in memory.

- We should also vectorize the labels, which is straightforward:

```
y_train = np.asarray(y_train).astype('float32')
y_test = np.asarray(y_test ).astype('float32')
```

Lab: IMDb Document Classification

- First we fit a `lasso logistic regression` model on the training data, and evaluate its performance on the test data.

```
from sklearn.linear_model import Lasso, LassoCV
from sklearn.metrics import mean_absolute_error, mean_squared_error
lasso = Lasso(max_iter=1000)
lassocv = LassoCV(alphas=None, cv=2, max_iter=1000)
#2-fold CV (you may try 5 or 10-fold) to choose alpha
lassocv.fit(x_train_1h, y_train)
```

```
## LassoCV(cv=2)
```

```
lasso.set_params(alpha=lassocv.alpha_)
```

```
## Lasso(alpha=0.00035649621258530 47)
```

```
lasso.fit(x_train_1h, y_train)
#mean_squared_error(y_test,lasso.predict(x_test_1h) )
```

```
## Lasso(alpha=0.00035649621258530 47)
```

```
mean_absolute_error(y_test,lasso.predict(x_test_1h) )
```

```
## 0.2869550774416491
```

Lab: IMDb Document Classification

- Next we fit a fully-connected neural network with two hidden layers, each with 16 units and ReLU activation.

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(16, activation='relu',input_dim=10000))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

print(model.summary())
```

```
## Model: "sequential_3"
## _________________________________________________________________
##  Layer (type)                 Output Shape              Param #
## =================================================================
##  dense_6 (Dense)              (None, 16)                160016
##
##  dense_7 (Dense)              (None, 16)                272
##
##  dense_8 (Dense)              (None, 1)                 17
##
## =================================================================
## Total params: 160,305
## Trainable params: 160,305
## Non-trainable params: 0
```

Lab: IMDb Document Classification

- The compilation step

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

- Train the model using epochs = 2

```
history=model.fit(x_train_1h, y_train, epochs=2,
batch_size=512, verbose=0,
#validation_split = 0.1
validation_data=(x_test_1h, y_test))
```

Lab: IMDb Document Classification

- Prediction on the test data

```
imdb_pred= model.predict(x_test_1h) #predicted values
```

```
##
##   1/782 [..............................] - ETA: 2:06
##  30/782 [>.............................] - ETA: 1s
##  65/782 [=>............................] - ETA: 1s
## 104/782 [==>...........................] - ETA: 1s
## 155/782 [====>.........................] - ETA: 0s
## 202/782 [======>.......................] - ETA: 0s
## 243/782 [========>.....................] - ETA: 0s
## 281/782 [=========>....................] - ETA: 0s
## 304/782 [==========>...................] - ETA: 0s
## 325/782 [===========>..................] - ETA: 0s
## 346/782 [============>.................] - ETA: 0s
## 374/782 [=============>................] - ETA: 0s
## 402/782 [==============>...............] - ETA: 0s
## 426/782 [===============>..............] - ETA: 0s
## 448/782 [================>.............] - ETA: 0s
## 473/782 [=================>............] - ETA: 0s
## 492/782 [=================>............] - ETA: 0s
## 514/782 [==================>...........] - ETA: 0s
## 538/782 [===================>..........] - ETA: 0s
## 559/782 [====================>.........] - ETA: 0s
## 584/782 [=====================>........] - ETA: 0s
```

Lab: IMDb Document Classification

- It seems NN has better predictions

```
np.mean(abs(y_test- imdb_pred[:,0]))
```

```
## 0.20150635
```

- The loss and accuracy of the network over the test data

```
test_loss, test_acc = model.evaluate(x_test_1h, y_test)
```

```
##
##   1/782 [..............................] - ETA: 51s - loss: 0.2889 - accuracy: 0.
##  16/782 [..............................] - ETA: 2s - loss: 0.2802 - accuracy: 0.8
##  33/782 [>.............................] - ETA: 2s - loss: 0.2771 - accuracy: 0.8
##  51/782 [>.............................] - ETA: 2s - loss: 0.2841 - accuracy: 0.8
##  71/782 [=>............................] - ETA: 2s - loss: 0.2917 - accuracy: 0.8
##  89/782 [==>...........................] - ETA: 2s - loss: 0.2952 - accuracy: 0.8
## 111/782 [===>..........................] - ETA: 1s - loss: 0.2961 - accuracy: 0.8
## 133/782 [====>.........................] - ETA: 1s - loss: 0.3008 - accuracy: 0.8
## 153/782 [====>.........................] - ETA: 1s - loss: 0.2999 - accuracy: 0.8
## 162/782 [=====>........................] - ETA: 2s - loss: 0.3002 - accuracy: 0.8
## 168/782 [=====>........................] - ETA: 2s - loss: 0.3000 - accuracy: 0.8
## 177/782 [=====>........................] - ETA: 2s - loss: 0.2994 - accuracy: 0.8
## 197/782 [======>.......................] - ETA: 2s - loss: 0.3011 - accuracy: 0.8
## 218/782 [=======>......................] - ETA: 2s - loss: 0.3011 - accuracy: 0.8
## 243/782 [========>.....................] - ETA: 2s - loss: 0.3008 - accuracy: 0.8
```

Lab: IMDb Document Classification

```
print('test_loss:', test_loss)
```

```
## test_loss: 0.3028479814529419
```

```
print('test_acc:', test_acc)
```

```
## test_acc: 0.8795199990272522
```

Convolution

- CNN (Convolutional Neural Networks) is mainly used for processing image or pixel data.
- Let's see the mathematical definition of Convolution first.

Definition: Convolution of discrete functions

If f and g are discrete functions, then $f * g$ is the convolution of f and g and is defined as:

$$(f * g)(x) = \sum_{u=-\infty}^{\infty} f(u)g(x-u)$$

- Intuitively, the convolution of two functions represents the amount of overlap between the two functions. The function f is called the *input*, g the *kernel* of the convolution.

commutativity

$f * g = g * f = \sum_{u=-\infty}^{\infty} g(u)f(x-u)$

- It doesn't matter if g or f is the kernel, due to commutativity.

Convolution

- To better understand convolution, we compute the convolution of f and g manually.
- Convolution involves a shifting of one function over another. The result is the overlap that occurs as one function is shifted over the other.
- Example: Suppose there are two function f and g shown below:

u	0	1	2	3
f(u)	1	2	3	4

u	1	2
g(u)	1	2

- First, **flip** function $g(u)$ to $g(-u)$

-u	-2	-1
g(-u)	2	1

- Then, **shift** $g(-u)$ by x unit.

x-u	x-2	x-1
g(x-u)	2	1

Convolution

- To find $f * g(1)$, we can arrange the values of the two functions as follows

$$\begin{array}{cccccc} & 1 & 2 & 3 & 4 \\ 2 & 1 & & & \end{array}$$

So we get $f * g(1) = 0 \cdot 2 + 1 \cdot 1 = 1$.

- To get find $f * g(2)$, we can arrange the values of the two functions as follows

$$\begin{array}{cccccc} 1 & 2 & 3 & 4 \\ 2 & 1 & & \end{array}$$

So we get $f * g(2) = 1 \cdot 2 + 2 \cdot 1 = 4$.

Keep doing the sliding, we get

x	1	2	3	4	5
(f*g)(x)	1	4	7	10	8

Convolution

If a function f ranges over a finite set of values, eg. $1, 2, \ldots, m$, then it can be represented as vector $\mathbf{a} = [a_1, a_2, \ldots, a_m]$.

Definition: Convolution of Vectors

If there are two vectors: $\mathbf{a} = [a_1, a_2, \ldots, a_m]$ and $\mathbf{b} = [b_1, b_2, \ldots, b_n]$, then the convolution $f * g$ is a vector $\mathbf{c} = [c_1, c_2, \ldots, c_{m+n-1}]$ with

$$c_j = \sum_u a_u b_{j-u+1},$$

where u ranges over all legal subscripts.

- As seen from the above example, we can think of this operation is that we're sliding the kernel over the input image(vector, 1-dimension): For each position of the kernel, we multiply the overlapping values of the kernel and image together, and add up the results.

Convolution

- Example: $\mathbf{a} = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$. Then

$$\mathbf{a} * \mathbf{b} = \begin{bmatrix} a_1 b_1 \\ a_1 b_2 + a_2 b_1 \\ a_1 b_3 + a_2 b_2 + a_3 b_1 \\ a_2 b_3 + a_3 b_2 \\ a_3 b_3 \end{bmatrix} = \begin{bmatrix} 0 \cdot 1 \\ 1 \cdot 1 + 0 \cdot 0 \\ 2 \cdot 1 + 1 \cdot 0 + 0 \cdot (-1) \\ 2 \cdot 0 + 1 \cdot (-1) \\ 2 \cdot (-1) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ -1 \\ -2 \end{bmatrix}$$

Convolution

- We can extend convolution to functions of two variables $f(x, y)$ and $g(x, y)$.

Definition: Convolution for Functions of two Variables

If f and g are discrete functions of two variables, then the convolution of f and g, $f * g$ is defined as:

$$(f * g)(x, y) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f(u, v) g(x - u, y - v)$$

Convolution

- We can regard functions of two variables as matrices with $A_{xy} = f(x, y)$, and obtain a matrix definition of convolution.

Definition: Convolution of Matrices

The convolution of an $m \times n$ matrix A and a $p \times q$ matrix B is an $(m + p - 1) \times (n + q - 1)$ matrix C with with

$$c_{xy} = \sum_u \sum_v a_{uv} b_{x-u+1, y-v=1},$$

where u and v ranges over all legal subscripts.

- We can also define convolution for continuous functions. In this case, we replace the sums by integrals in the definitions.

A Convolution Example

- Example (https://www.allaboutcircuits.com/technical-articles/two-dimensional-convolution-in-image-processing/)

Input matrices, where x represents the original image and h represents the kernel:

25	100	75	49	130
50	80	0	70	100
5	10	20	30	0
60	50	12	24	32
37	53	55	21	90
140	17	0	23	222

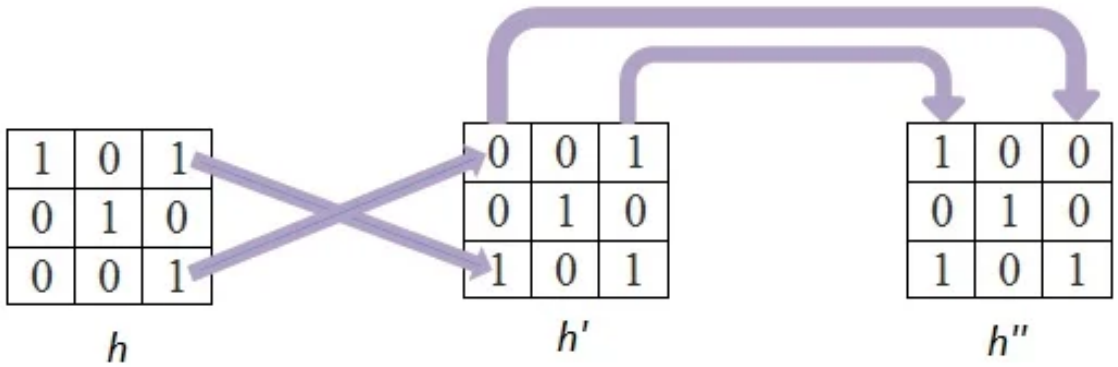
x

1	0	1
0	1	0
0	0	1

h

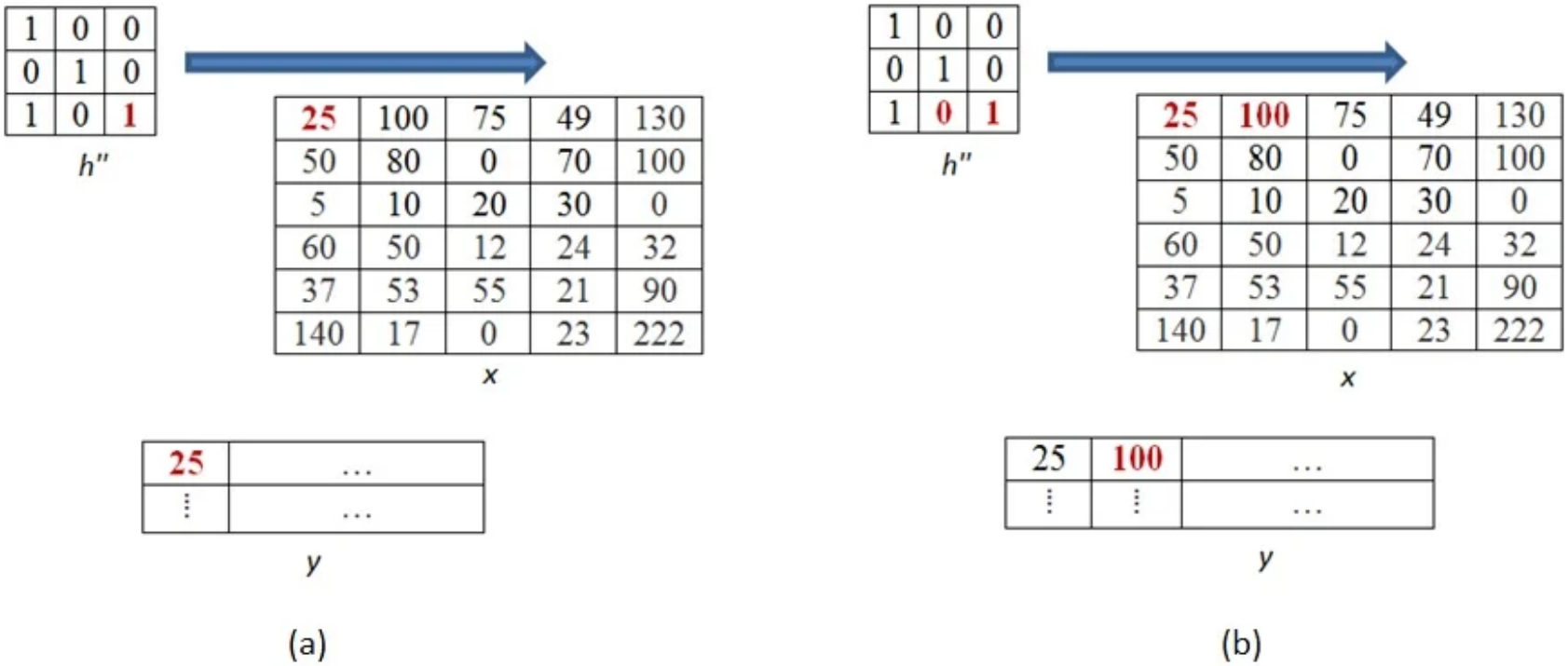
A Convolution Example

- Step 1: Matrix inversion
  - This step involves flipping of the kernel along, say, rows followed by a flip along its columns, as shown below

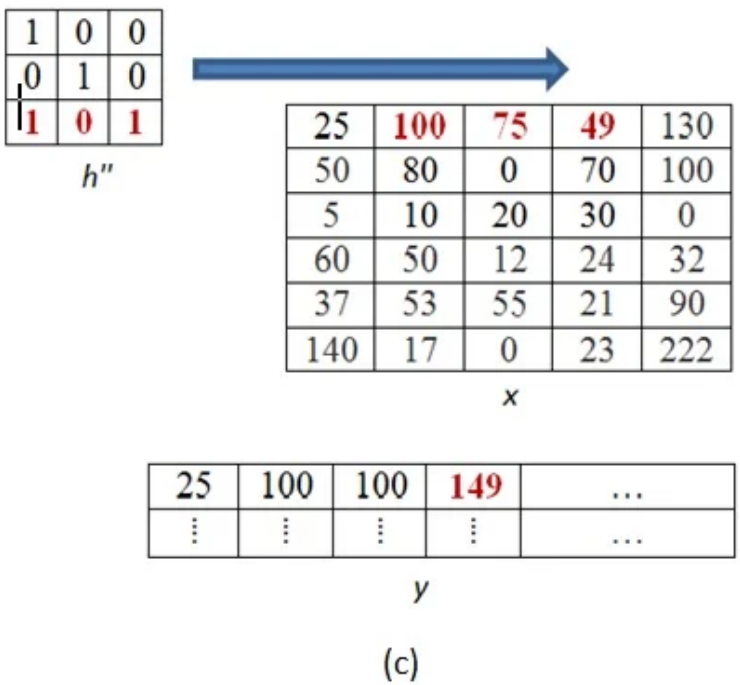


A Convolution Example

- Step 2: Slide the kernel over the image and perform MAC operation at each instant: Overlap the inverted kernel over the image, advancing pixel-by-pixel.
  - For each case, compute the product of the mutually overlapping pixels and calculate their sum. The result will be the value of the output pixel at that particular location. Non-overlapping pixels will be assumed to have a value of 0.



A Convolution Example



1	0	0
0	1	0
1	0	1

h''

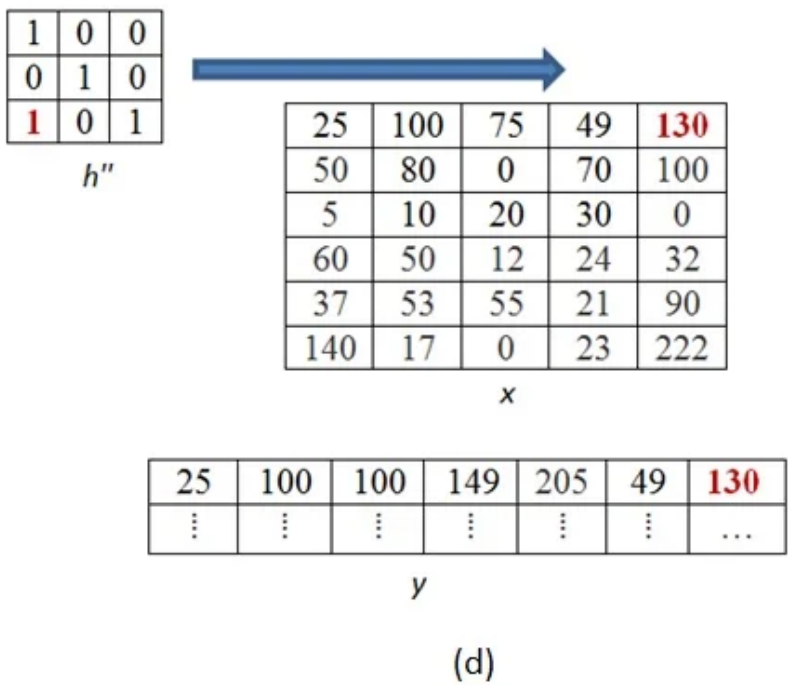
25	100	75	49	130
50	80	0	70	100
5	10	20	30	0
60	50	12	24	32
37	53	55	21	90
140	17	0	23	222

x

25	100	100	149	…
⋮	⋮	⋮	⋮	…

y

(c)



1	0	0
0	1	0
1	0	1

h''

25	100	75	49	130
50	80	0	70	100
5	10	20	30	0
60	50	12	24	32
37	53	55	21	90
140	17	0	23	222

x

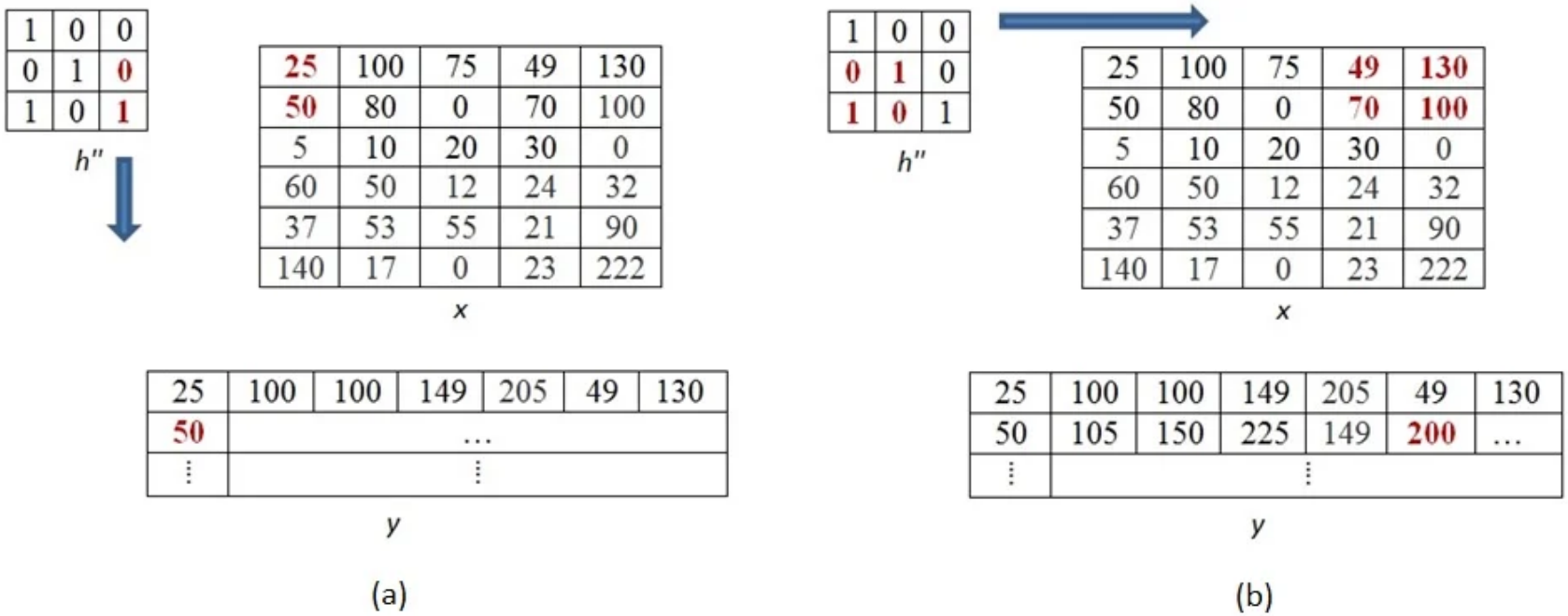
25	100	100	149	205	49	130
⋮	⋮	⋮	⋮	⋮	⋮	…

y

(d)

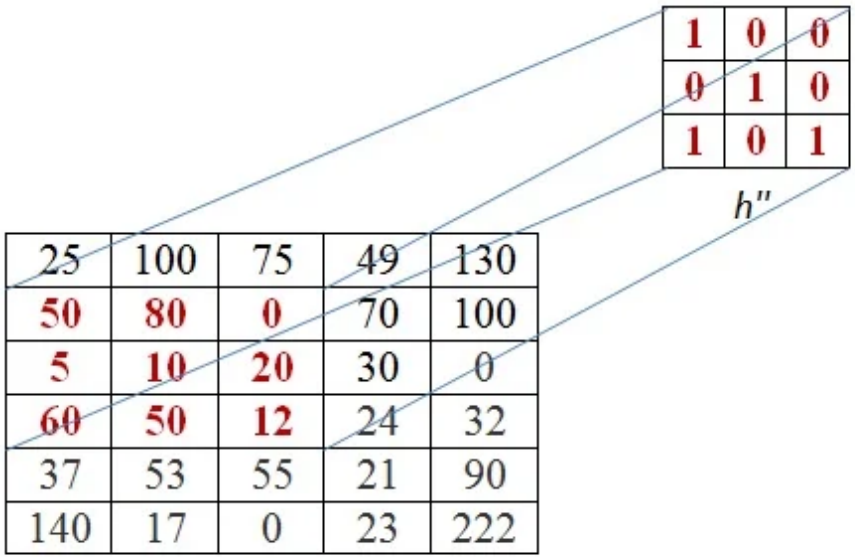
A Convolution Example

- Move Down Vertically, Advance Horizontally



A Convolution Example

- This process of moving one step down followed by horizontal scanning has to be continued until the last row of the image matrix.



25	100	100	149	205	49	130
50	105	150	225	149	200	100
5	60	130	140	165	179	130
60	55	132	...			
⋮	⋮					

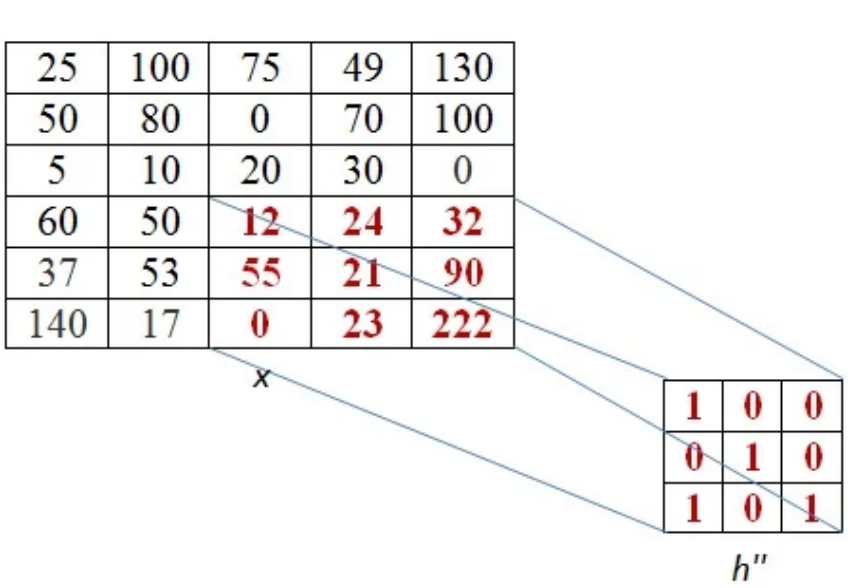
y

$$y(4,3) = 50 \times 1 + 80 \times 0 + 0 \times 0 + 5 \times 0 + 10 \times 1 + 20 \times 0 + 60 \times 1 + 50 \times 0 + 12 \times 1$$

$$= 50 + 0 + 0 + 0 + 10 + 0 + 60 + 0 + 12 = 132$$

A Convolution Example

- This process of moving one step down followed by horizontal scanning has to be continued until the last row of the image matrix.



25	100	100	149	205	49	130
50	105	150	225	149	200	100
5	60	130	140	165	179	130
60	55	132	174	74	94	132
37	113	147	96	189	83	90
140	54	253	145	255	…	
⋮	⋮					

y

$$y(6,5) = 12 \times 1 + 24 \times 0 + 32 \times 0 + 55 \times 0 + 21 \times 1 + 90 \times 0 + 0 \times 1 + 23 \times 0 + 222 \times 1$$

$$= 12 + 0 + 0 + 0 + 21 + 0 + 0 + 0 + 222 = 255$$

A Convolution Example

- This process of moving one step down followed by horizontal scanning has to be continued until the last row of the image matrix.

25	100	75	49	130
50	80	0	70	100
5	10	20	30	0
60	50	12	24	32
37	53	55	21	90
140	17	0	**23**	**222**

x

1	**0**	0
0	1	0
1	0	1

h''

25	100	100	149	205	49	130
50	105	150	225	149	200	100
5	60	130	140	165	179	130
60	55	132	174	74	94	132
37	113	147	96	189	83	90
140	54	253	145	255	137	254
0	140	54	53	78	243	90
0	0	140	17	0	**23**	…

y

$$y(8, 6) = 23 \times 1 + 222 \times 0 = 23 + 0 = 23$$

A Convolution Example

- The resultant output matrix

25	100	100	149	205	49	130
50	105	150	225	149	200	100
5	60	130	140	165	179	130
60	55	132	174	74	94	132
37	113	147	96	189	83	90
140	54	253	145	255	137	254
0	140	54	53	78	243	90
0	0	140	17	0	23	255

y

A Convolution Example

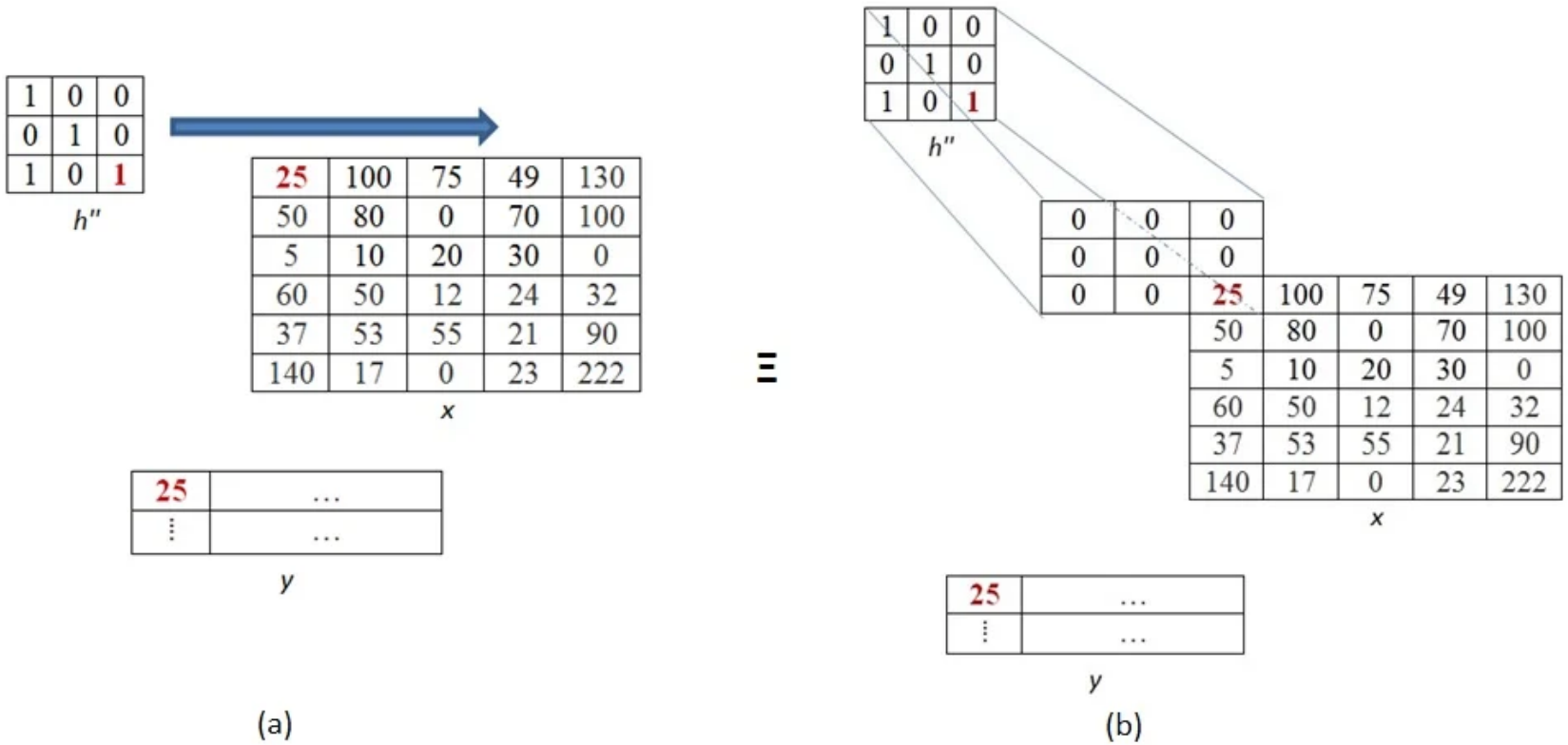
- The result is actually obtained using **Zero Padding** - adding zeros around the edges of the input.
- The zero padding matrix used here is:

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	25	100	75	49	130	0	0
0	0	50	80	0	70	100	0	0
0	0	5	10	20	30	0	0	0
0	0	60	50	12	24	32	0	0
0	0	37	53	55	21	90	0	0
0	0	140	17	0	23	222	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

x

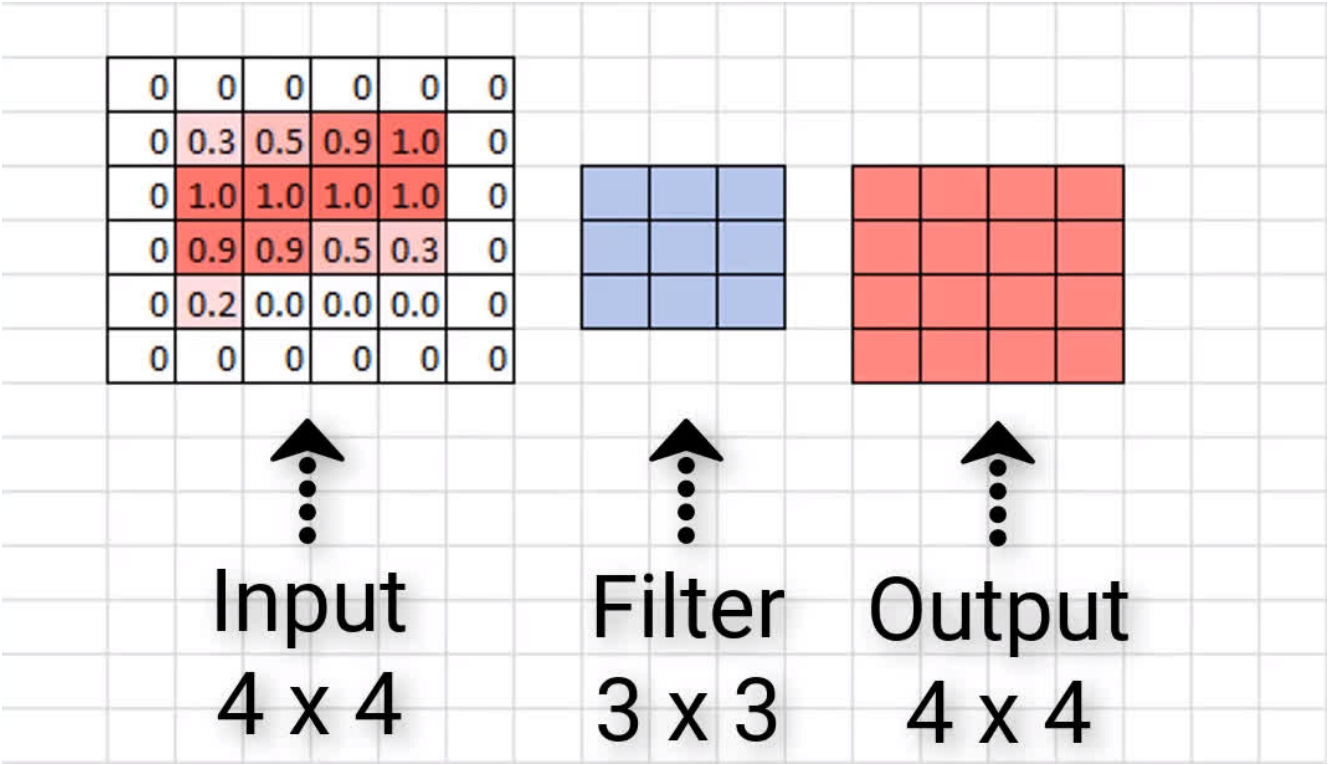
A Convolution Example

- For example,



A Convolution Example

- **Note**: Images are stored in the form of a matrix of numbers in a computer where these numbers are known as pixel values. In the application of convolution to image processing, we compute only same/smaller area as input has been defined.
- For the output layer to have the same input size as the input layer, we need the zero padding matrix with only one layer of zeros.



A Convolution Example

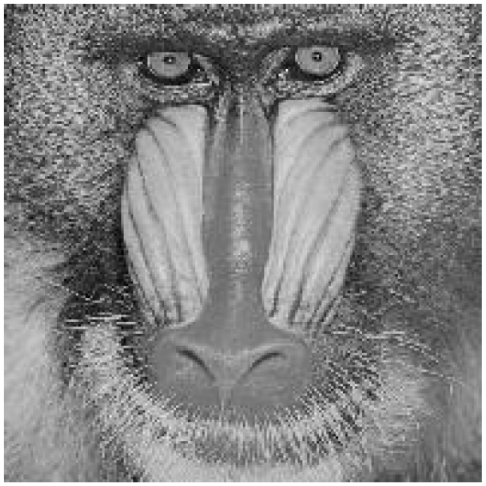
- In the above example, for the output layer to have size 6×5 (the input size), we need the following zero padding matrix.

0	0	0	0	0	0	0
0	25	100	75	49	130	0
0	50	80	0	70	100	0
0	5	10	20	30	0	0
0	60	50	12	24	32	0
0	37	53	55	21	90	0
0	140	17	0	23	222	0
0	0	0	0	0	0	0

x

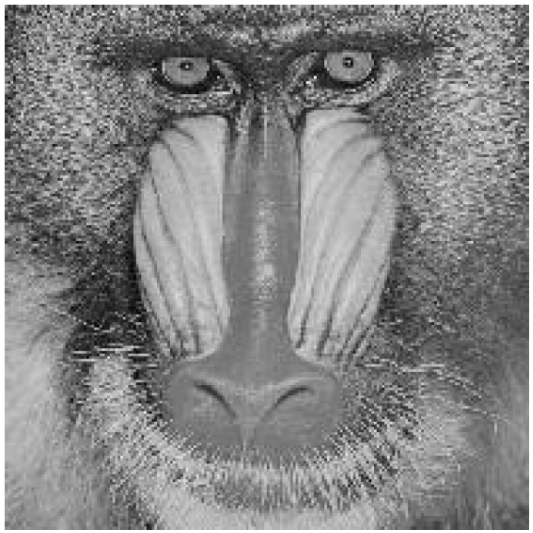
Convolution

- Convolving an image with a kernel (typically a 3×3 matrix) is a powerful tool for image processing.
- Image filtering: Replacing each pixel value by weighted average of its neighbors

$B =$  $K * B =$ 

$K = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$ implements a *mean filter* which smooths an image by replacing each pixel value with the mean of its neighbors.

Convolution

$B =$  $K * B =$ 

The kernel $K = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$ implements the *Sobel edge detector*. It detects gradients in the pixel values (sudden changes in brightness), which correspond to edges. The example is for vertical edges.

Convolution

- Let's see an application of convoluting an image
  - we use library `scikit-image` import an image
  - We use `ndimage` in `Scipy` conduct convolutions
  - https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.convolve.html

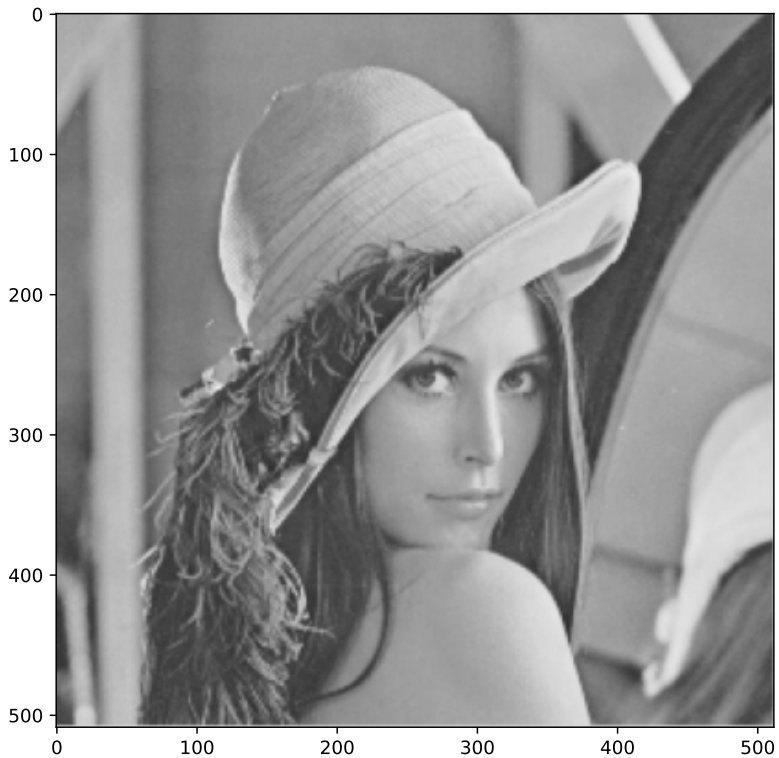
```
# install scikit-image first
from skimage import color
from skimage import io
import numpy as np
from scipy import ndimage
import matplotlib.pyplot as plt
```

Convolution

```
img_raw = io.imread('../data/woman.png')
print(img_raw.ndim)
```

```
## 3
```

```
plt.imshow(img_raw)
plt.show()
```



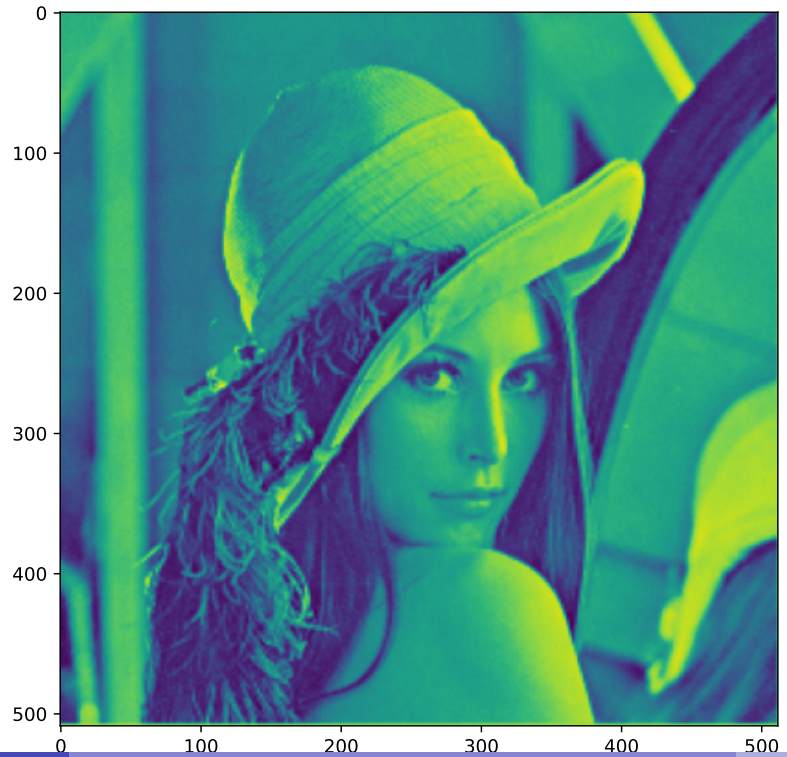
Convolution

- The image has 3 channels,but they are identical

```
print(np.sum(abs(img_raw[:,:,0]-img_raw[:,:,1])))
```

```
## 0
```

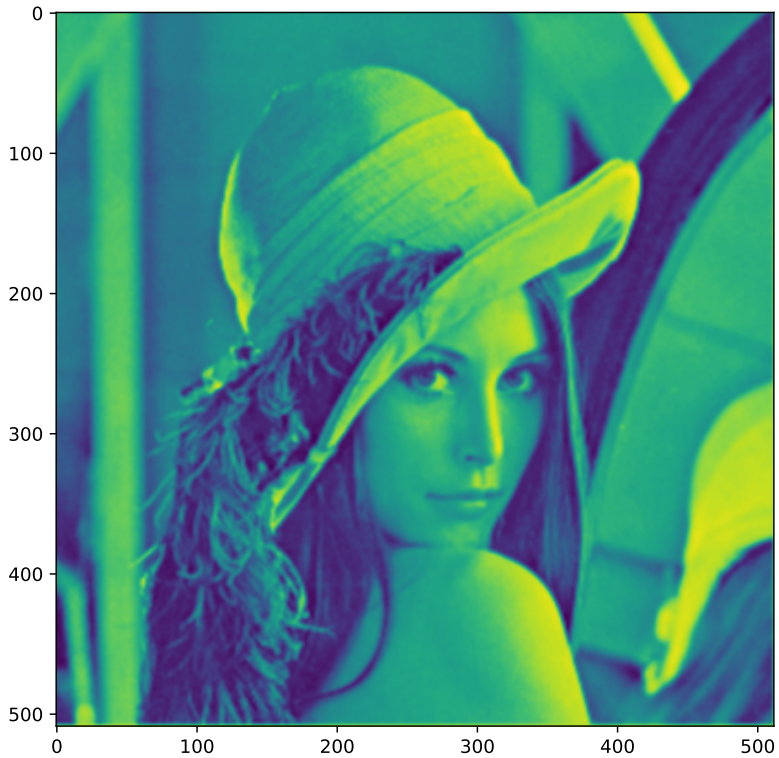
```
img_gray= img_raw[:,:,0] # or: img_gray = color.rgb2gray(img_raw)
plt.imshow(img_gray)
plt.show()
```



Convolution

- Smoothing the image using a mean filter

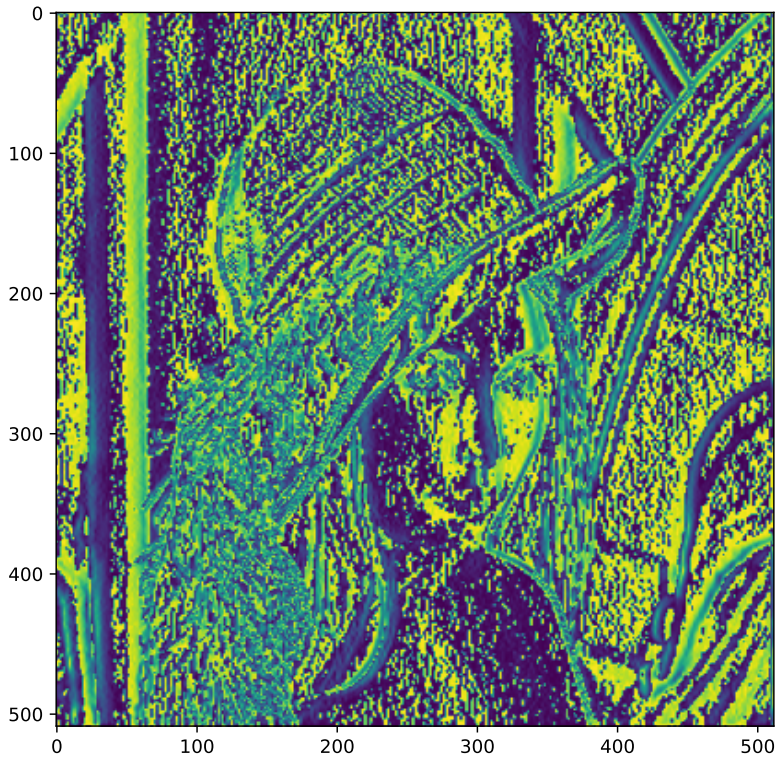
```
mean_filter=np.ones((3,3))/9
img2=ndimage.convolve(img_gray, mean_filter)
plt.imshow(img2)
plt.show()
```



Convolution

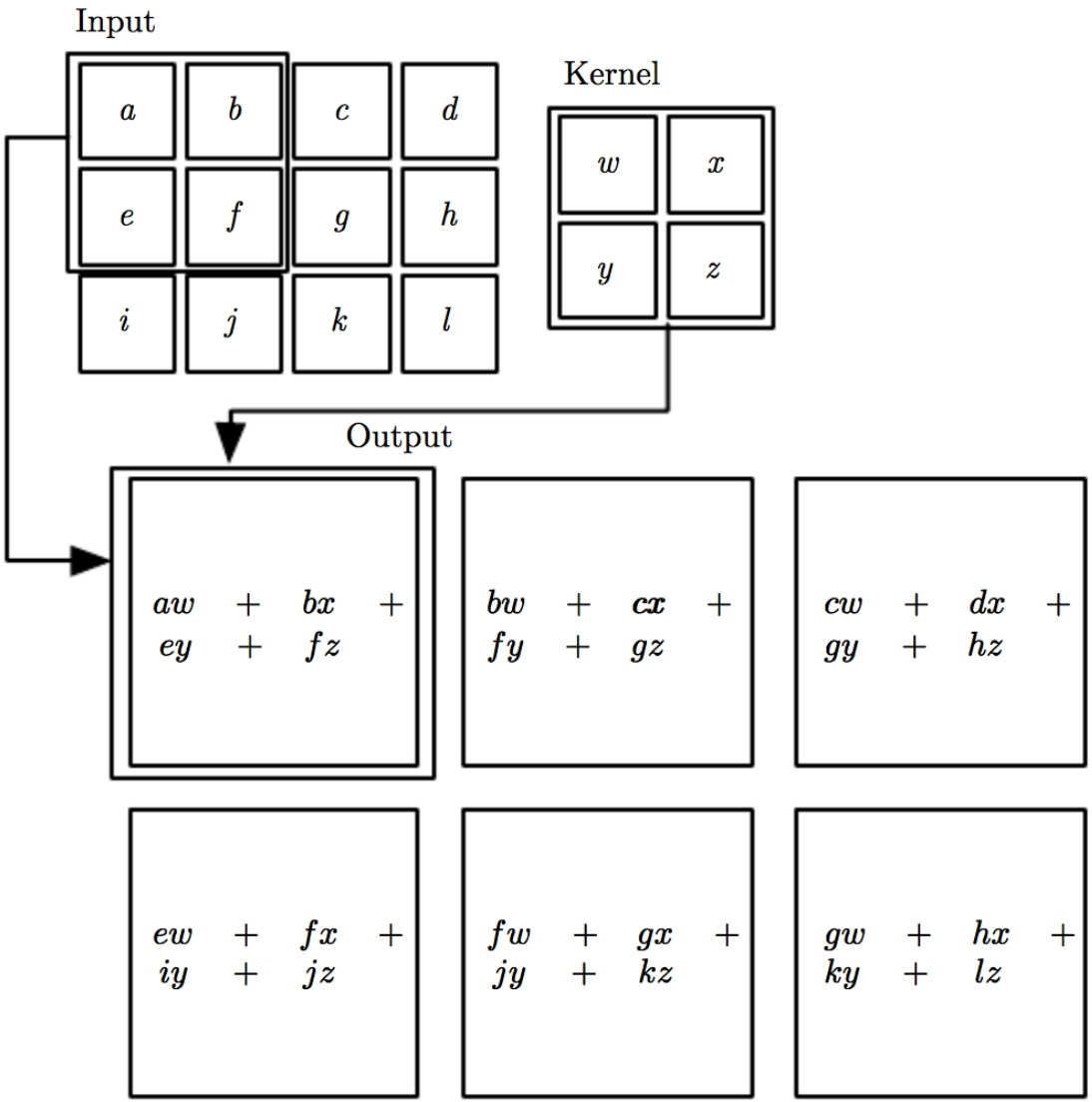
- Sobel edge detection

```
edge_filter=np.array([[1,0,-1],[2,0,-2],[1,0,-1]])
img3=ndimage.convolve(img_gray, edge_filter)
plt.imshow(img3)
plt.show()
```



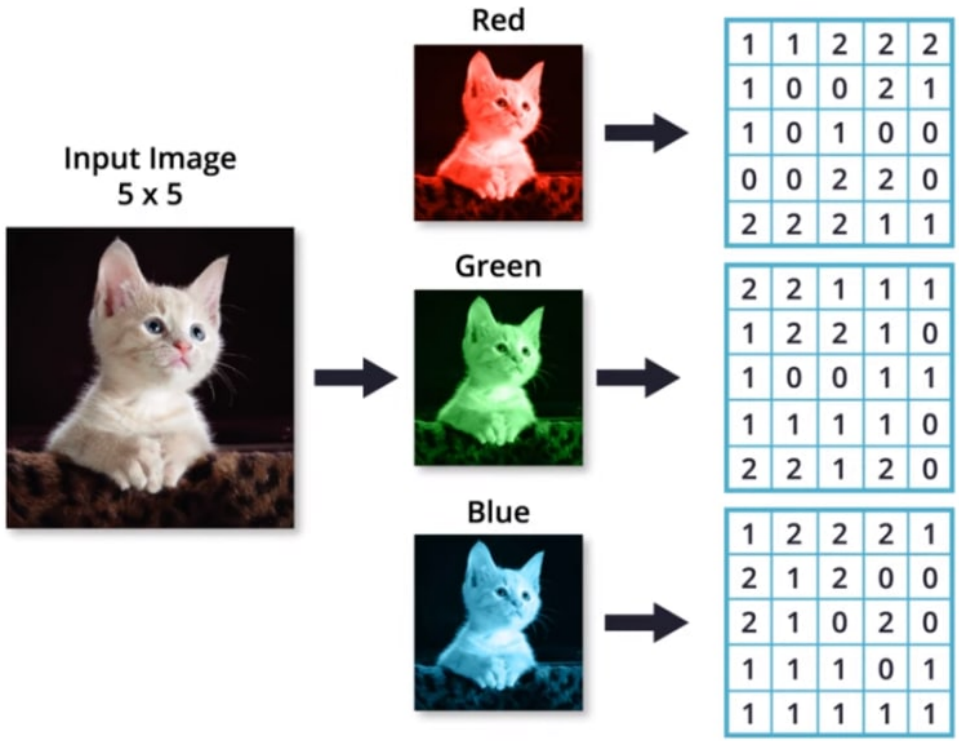
Convolution

- Convolution without kernel flipping (Cross-Correlation):
  - No zero-padding, output has decreases size



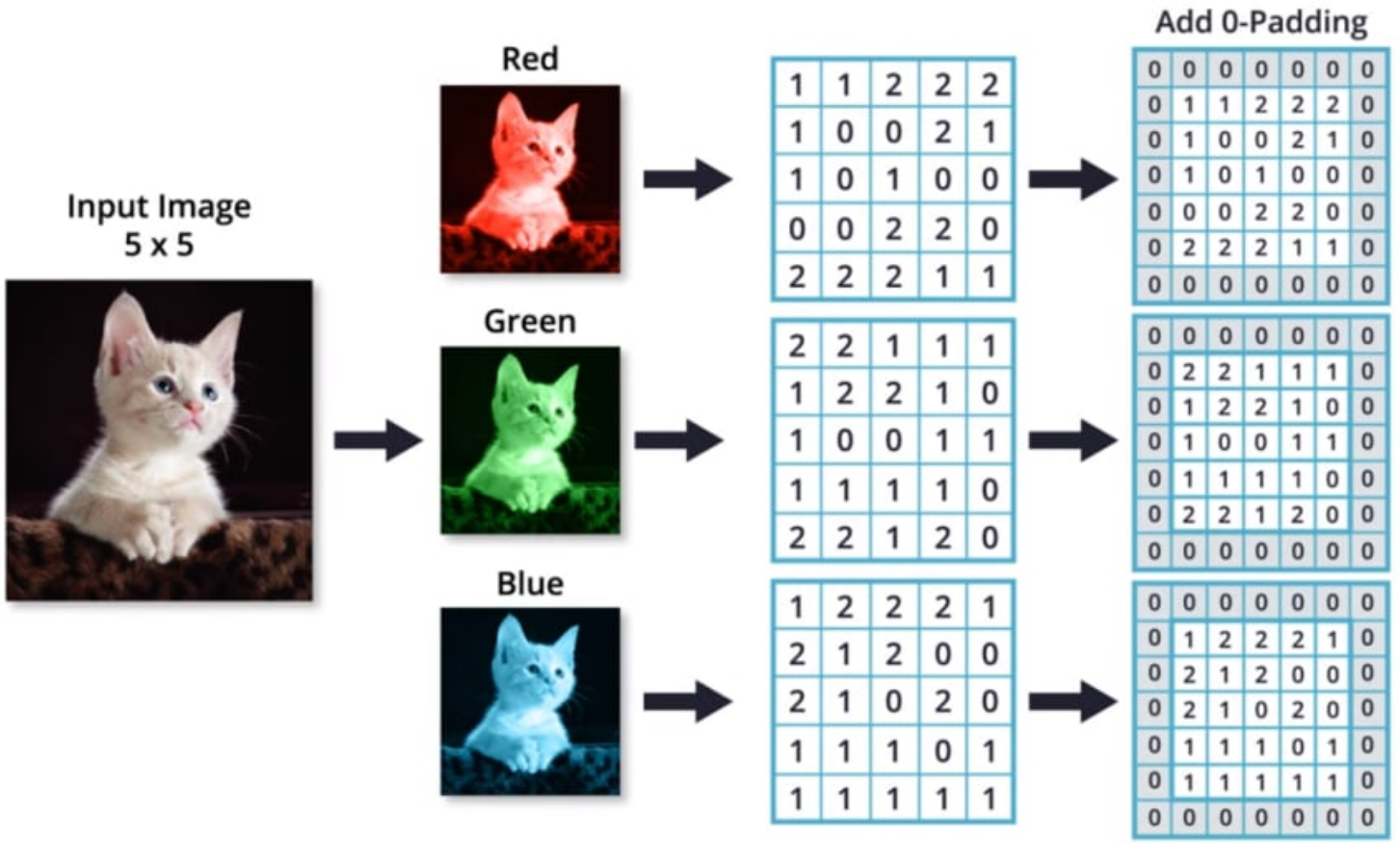
Convolution

- Similarly, we can define Convolution for functions of three variables, and a 3-variable function can be represented as a cube - combination of several matrices.
- For example, any color image has three channels, i.e. red, green, and blue. The following input is a feature map (array) with 3 channels, width 5 and height 5 (generally will be larger, like 32×32). The example is from https://dev.to/sandeepbalachandran/machine-learning-convolution-with-color-images-2p41.



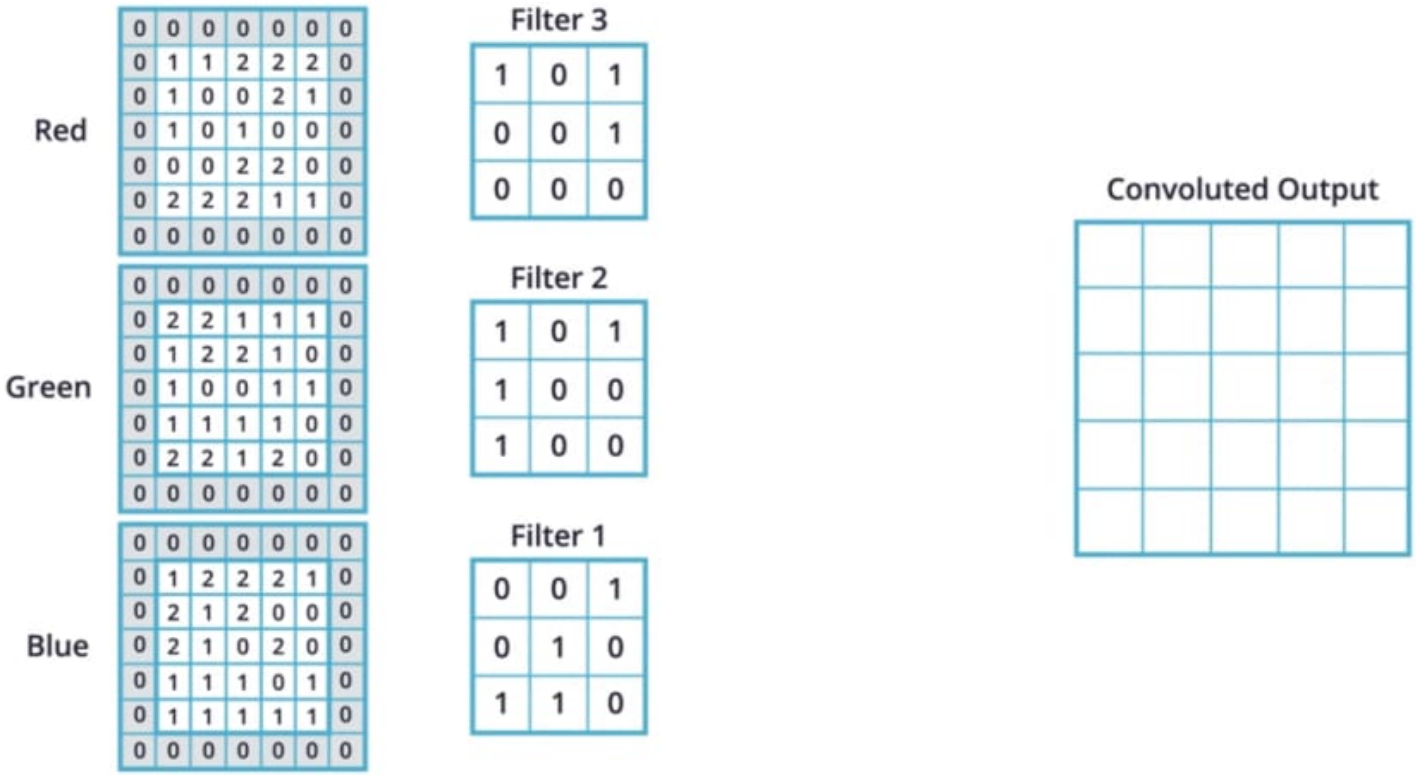
Convolution

- To get the output of the same size, we add zero padding to each of these arrays when performing the convolution.



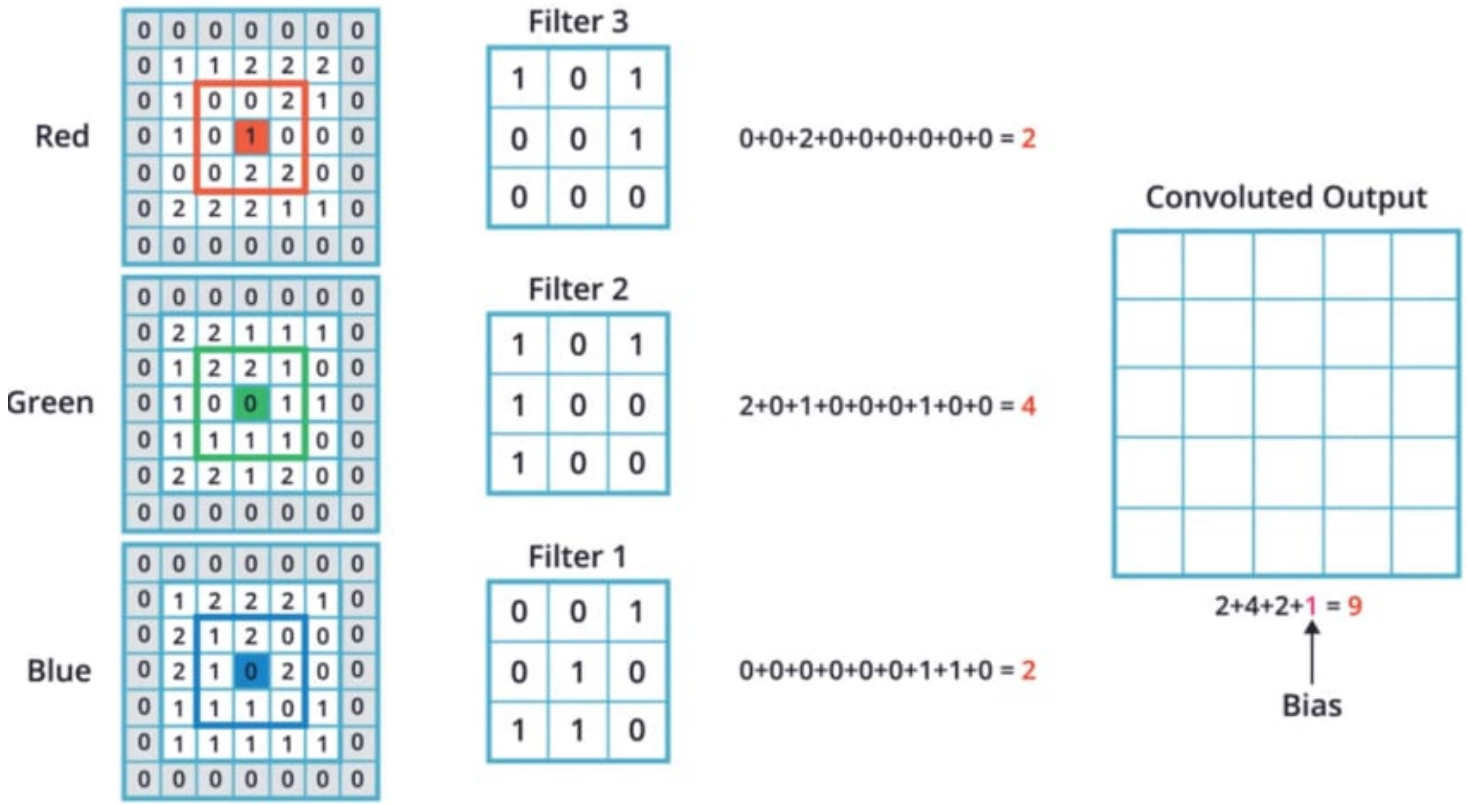
Convolution

- We need three filters to conduct convolution.
  - The three filters can be the same.



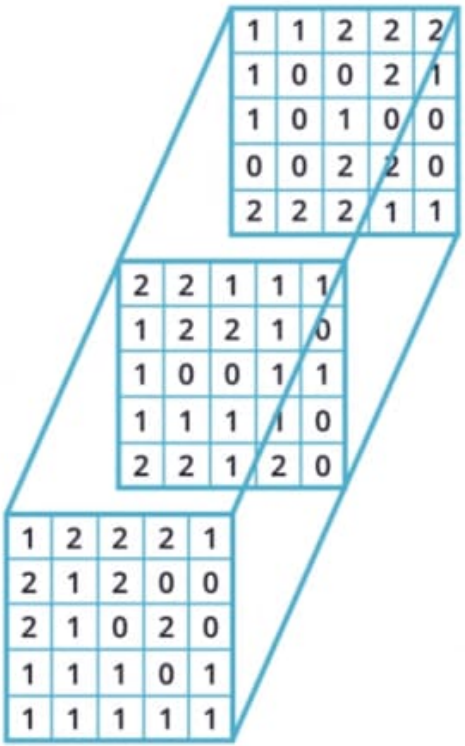
Convolution

- After calculating the convolutions for each color channel, we'll add these numbers together. However, it's customary to add a bias value, which usually has a value of 1.



Convolution

- A convolution with a single 3D (one per color) filter produces a single convoluted output. Notice that the color information has been used.



3D Filter

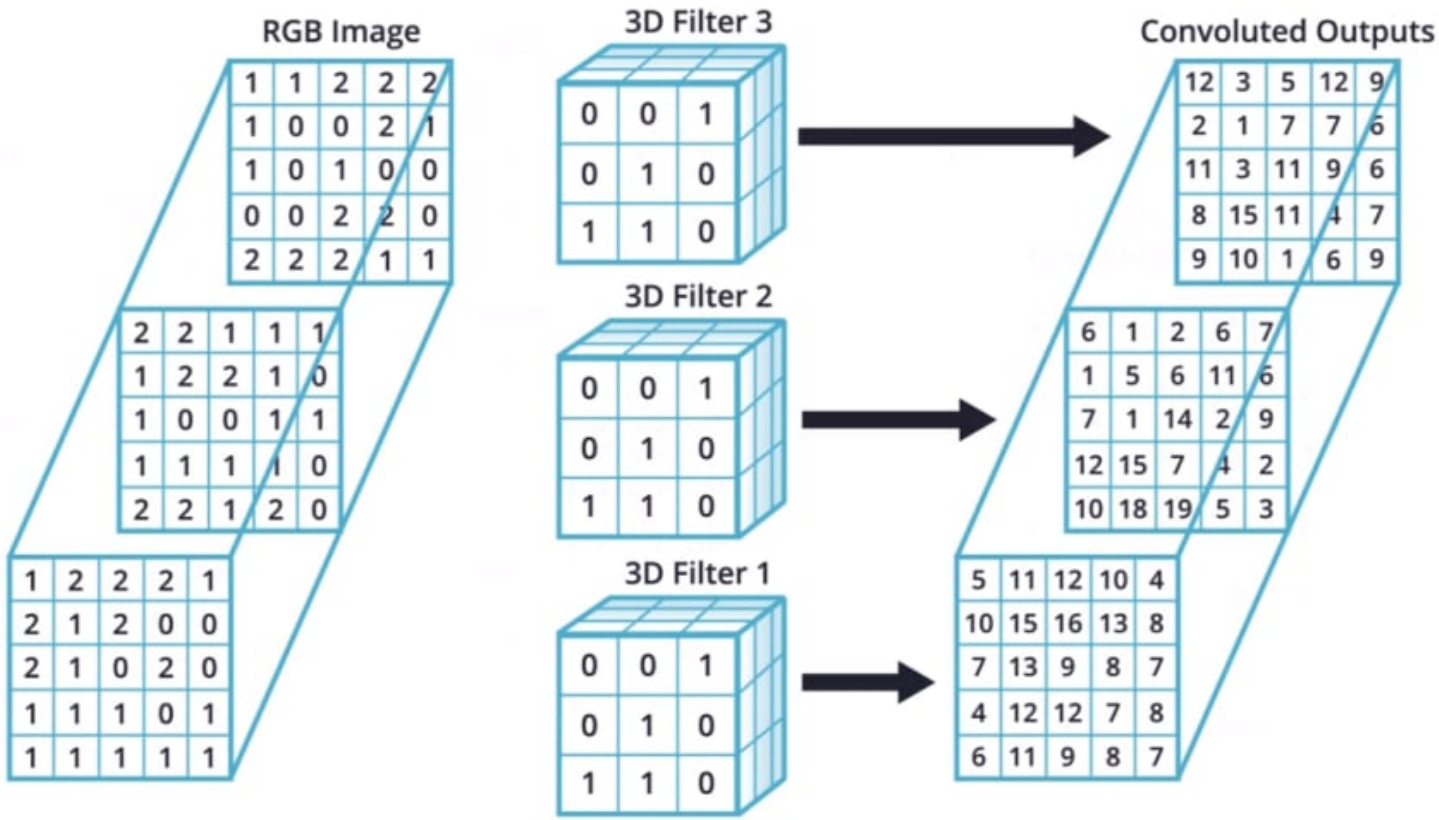
0	0	1
0	1	0
1	1	0

Convoluted Output

5	11	12	10	4
10	15	16	13	8
7	13	9	8	7
4	12	12	7	8
6	11	9	8	7

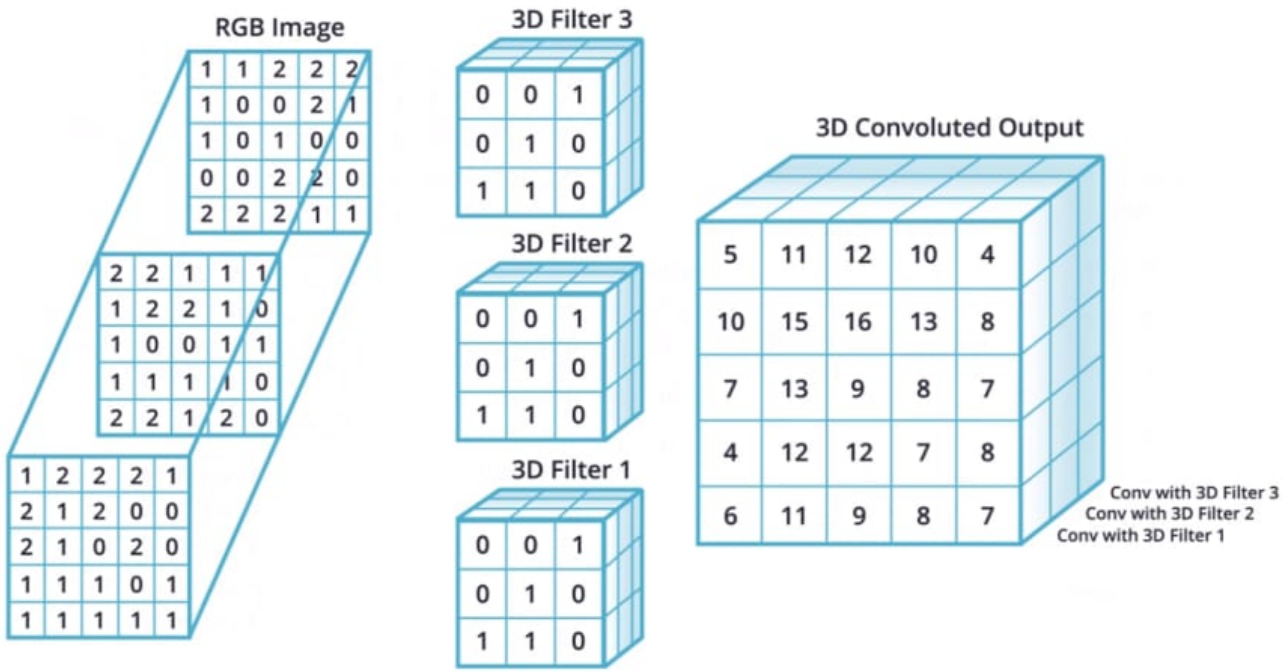
Convolution

- However, when working with CNNs, it's customary to use more than 1 3D filter.



Convolution

- Therefore, we can think of the convoluted output as being 3D, where the depth will correspond to the number of filters.
- If we use K different convolution 3D filters, we get K two-dimensional output feature maps.
- In the following, the output is treated as a single 3D feature map.



Pooling layers

- A pooling layer provides a way to condense a large image into a smaller summary image.
- There are mainly two types of pooling operations used in CNNs
  - 1. Max Pooling
  - 2. Average Pooling
- The max pooling operation summarizes each non-overlapping 2×2 block of pixels in an image using the maximum value in the block.

$$\text{Max pool} \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}$$

- This reduces the size of the image by a factor of two in each direction, and
- it also provides some *location invariance*: i.e. as long as there is a large value in one of the four pixels in the block, the whole block registers as a large value in the reduced image.

Pooling layers

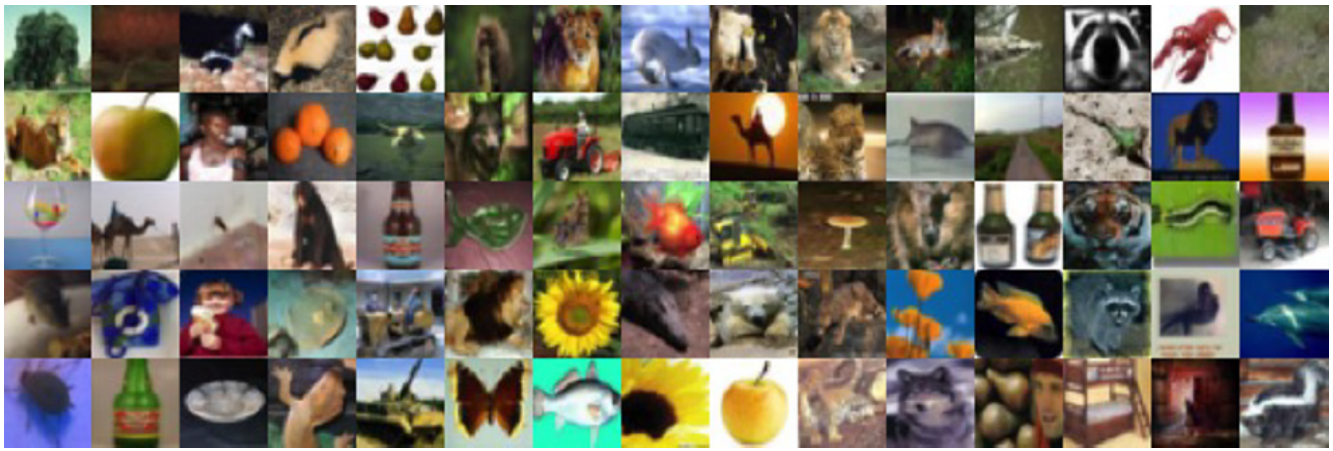
- Average Pooling: The average/mean value of all the pixels caught in the filter is returned to form an average pooled representation of the original image.

$$\text{Average pooling} \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 6/4 & 11/4 \\ 5/4 & 9/4 \end{bmatrix}$$

- It can be seen that pixel values are much larger in the max pooled representation compared to the average pooled representation.
- So Max Pooling sharpens the feature identification.

Convolutional Neural Networks

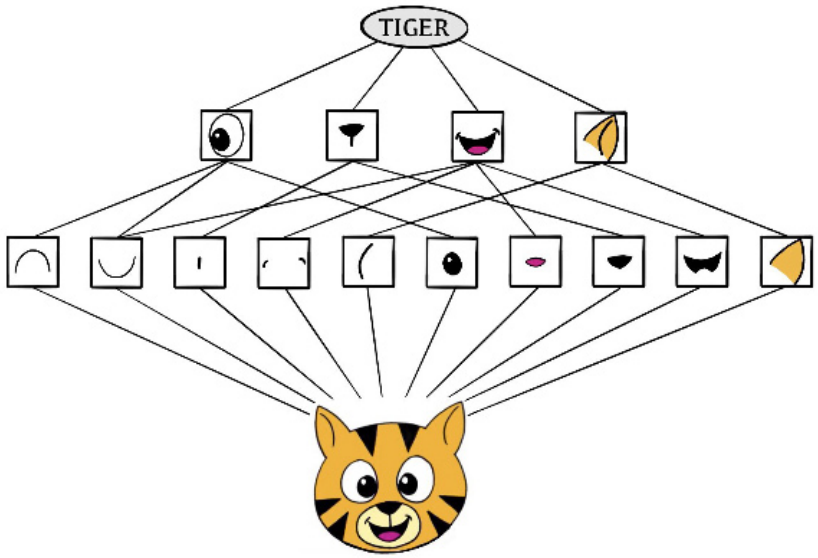
- One main application of CNN is classifying images.
- A sample of 32×32 images from the *CIFAR100* database: a collection of natural images from everyday life, with 100 different classes represented.
  - 50K training images, 10K test images.
  - Each image is a three-dimensional array or feature map (arrary): $32 \times 32 \times 3$ array of 8-bit numbers.



Convolutional Neural Networks

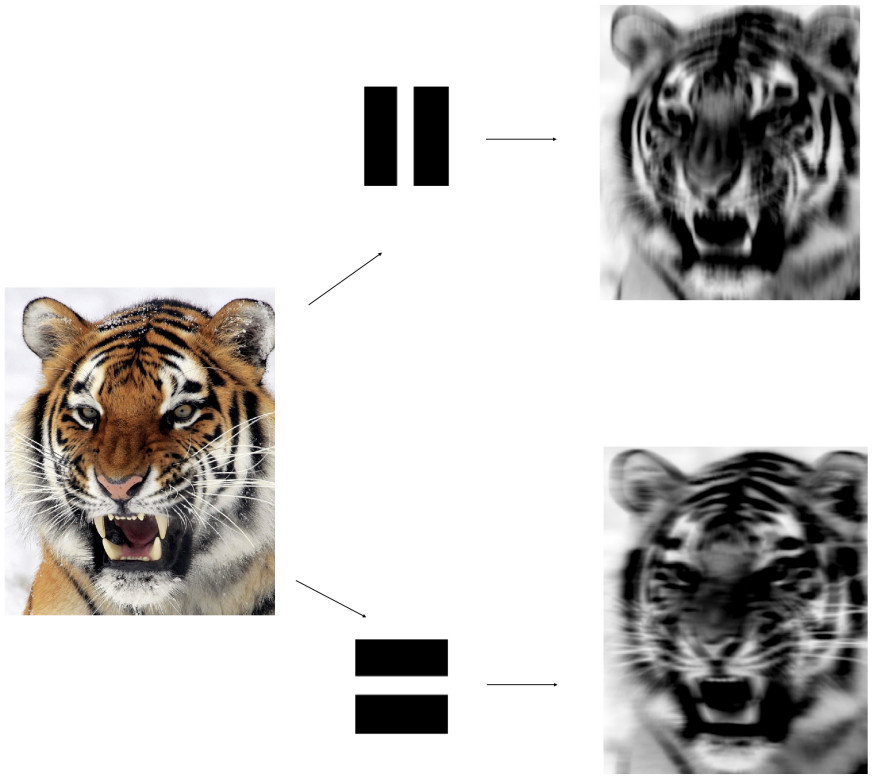
How CNN works?

- The CNN builds up an image in a hierarchical fashion.
  - Edges and shapes are recognized and pieced together to form more complex shapes, eventually assembling the target image.
- Figure (Schematic showing how a convolutional neural network classifies an image of a tiger):
  - The network first identifies low-level features in the input image, such as small edges, patches of color, and the like.
  - These low-level features are then combined to form higher-level features, such as parts of ears, eyes, and so on.
  - Eventually, the presence or absence of these higher-level features contributes to the probability of any given output class.



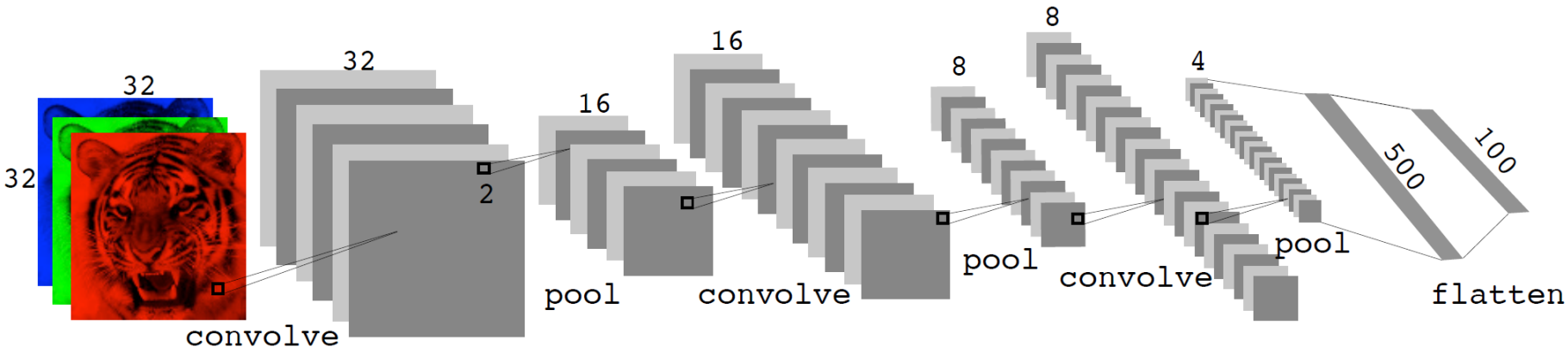
Convolutional Neural Networks

- This hierarchical construction is achieved using *convolution* and *pooling* layers.



- The idea of **convolution** with a filter is to find common patterns that occur in different parts of the image.
- The two filters shown here highlight vertical and horizontal stripes.
- The result of the convolution is a new feature map.

Architecture of a CNN



- Many convolve + pool layers.
- Filters are typically small, e.g. each channel 3×3.
- Each filter creates a new channel in convolution layer.
- As pooling reduces size, the number of filters/channels is typically increased.
- Number of layers can be very large. E.g. *resnet50* trained on imagenet 1000-class image data base has 50 layers!

Lab: CNN to the CIFAR data

- CIFAR100, a dataset of 50,000 training 32×32 RGB images belonging to 100 classes (50,000 images per class) and 10,000 test images.
  - https://keras.io/api/datasets/cifar100/

```
from tensorflow import keras
from keras.preprocessing import image
(x_train, g_train), (x_test, y_test) = keras.datasets.cifar100.load_data()
x_train.ndim
```

```
## 4
```

```
x_train.shape
```

```
## (50000, 32, 32, 3)
```

```
x_test.ndim
```

```
## 4
```

```
x_test.shape
```

```
## (10000, 32, 32, 3)
```

Lab: CNN to the CIFAR data

- The array of 50,000 training images has four dimensions: each three-color image is represented as a set of three channels, each of which consists of 32×32 eight-bit pixels. We standardize as we did for the digits, but keep the array structure. We one-hot encode the response factors to produce a 100-column binary matrix.
- Converting dataset labels to categorical
  - https://keras.io/api/utils/python_utils/#to_categorical-function

```
import numpy as np
from keras.utils import to_categorical
x_train = x_train/255
x_test = x_test/255
y_train = to_categorical(g_train, 100)
print(y_train.shape)
```

```
## (50000, 100)
```

Lab: CNN to the CIFAR data

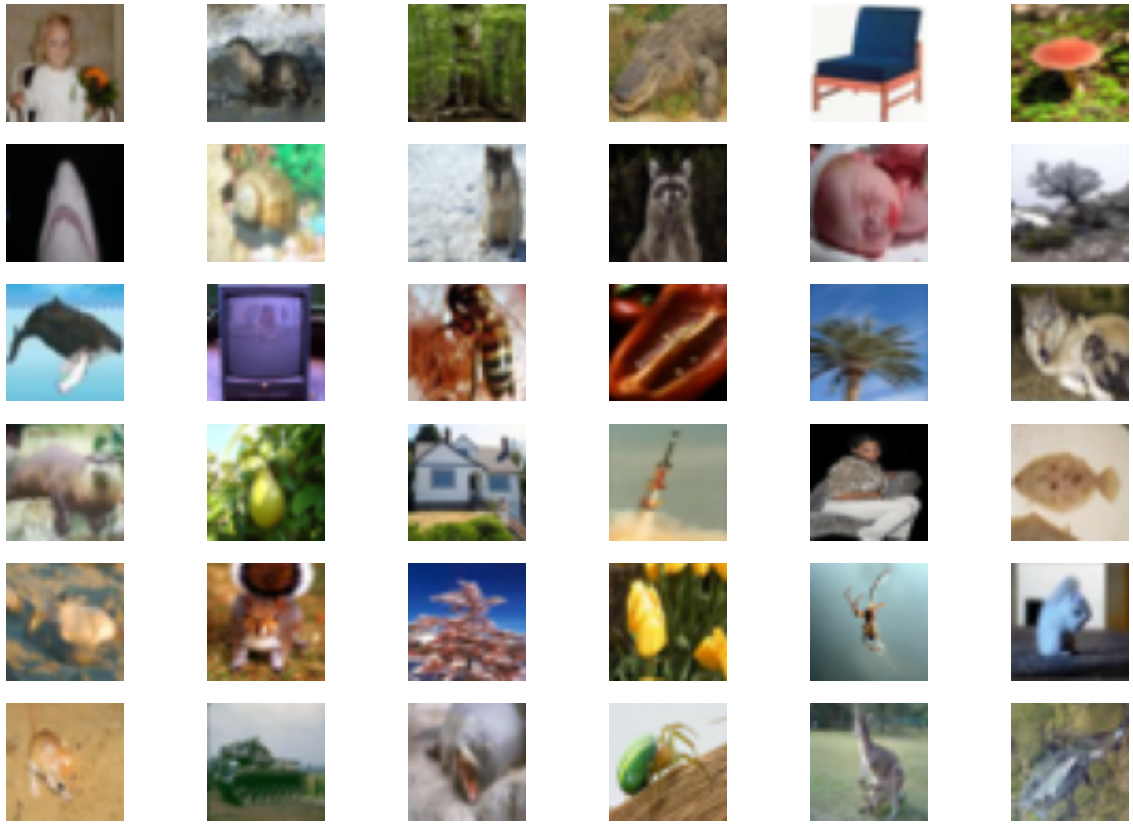
- Before we start, we look at some of the training images

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

#to get 6*6 = 36 images together
index=np.random.choice(50000, 36, replace=False)
fig, axes = plt.subplots(6, 6)
for i in range(6):
    for j in range(6):
      k = (i*6)+j
      axes[i,j].imshow(x_train[index[k]])
      axes[i,j].axis('off')
```

```
## <matplotlib.image.AxesImage object at 0x000001F461B6ABC0>
## (-0.5, 31.5, 31.5, -0.5)
## <matplotlib.image.AxesImage object at 0x000001F461B6B670>
## (-0.5, 31.5, 31.5, -0.5)
## <matplotlib.image.AxesImage object at 0x000001F461B6B340>
## (-0.5, 31.5, 31.5, -0.5)
## <matplotlib.image.AxesImage object at 0x000001F461B6AFE0>
## (-0.5, 31.5, 31.5, -0.5)
```

Lab: CNN to the CIFAR data

```
plt.show()
```



Lab: CNN to the CIFAR data

- Here we specify a moderately-sized CNN for demonstration purposes.
  - https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D
  - https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Conv2D(filters=32, kernel_size=(3, 3),
padding='same',activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(filters=64, kernel_size=(3, 3),
padding='same',activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(filters=128, kernel_size=(3, 3),
padding='same',activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(filters=256, kernel_size=(3, 3),
padding='same',activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
# Adding the fully connected layers to CNN:
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(100, activation='softmax'))
```

Lab: CNN to the CIFAR data

- Notice that we used the `padding = "same"` argument to `layers.Conv2D`, which ensures that the output channels have the same dimension as the input channels.
- There are 32 channels in the first hidden layer, in contrast to the three channels in the input layer. We use a 3×3 convolution filter for each channel in all the layers.
- Each convolution is followed by a max-pooling layer over 2×2 blocks.

Lab: CNN to the CIFAR data

```
print(model.summary())
```

```
## Model: "sequential_4"
## _________________________________________________________________
##  Layer (type)                 Output Shape              Param #
## =================================================================
##  conv2d (Conv2D)              (None, 32, 32, 32)        896
##
##  max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
##  )
##
##  conv2d_1 (Conv2D)            (None, 16, 16, 64)        18496
##
##  max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)         0
##  2D)
##
##  conv2d_2 (Conv2D)            (None, 8, 8, 128)         73856
##
##  max_pooling2d_2 (MaxPooling  (None, 4, 4, 128)        0
##  2D)
##
##  conv2d_3 (Conv2D)            (None, 4, 4, 256)         295168
##
##  max_pooling2d_3 (MaxPooling  (None, 2, 2, 256)        0
##  2D)
##
##  flatten (Flatten)            (None, 1024)              0
##
##  dropout_3 (Dropout)          (None, 1024)              0
##
##  dense_8 (Dense)              (None, 512)               524800
```

Lab: CNN to the CIFAR data

- Configuring Network for Training

```
model.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])
```

Lab: CNN to the CIFAR data

- Fit the CNN Model

```
history = model.fit(x_train, y_train, epochs=5,
batch_size=128, validation_split = 0.2)
```

```
## Epoch 1/5
##
##   1/313 [..............................] - ETA: 3:57 - loss: 4.6053 - ac
##   2/313 [..............................] - ETA: 34s - loss: 4.6051 - acc
##   3/313 [..............................] - ETA: 34s - loss: 4.6062 - acc
##   4/313 [..............................] - ETA: 35s - loss: 4.6054 - acc
##   5/313 [..............................] - ETA: 35s - loss: 4.6037 - acc
##   6/313 [..............................] - ETA: 35s - loss: 4.6083 - acc
##   7/313 [..............................] - ETA: 36s - loss: 4.6071 - acc
##   8/313 [..............................] - ETA: 36s - loss: 4.6069 - acc
##   9/313 [..............................] - ETA: 36s - loss: 4.6086 - acc
##  10/313 [..............................] - ETA: 36s - loss: 4.6086 - acc
##  11/313 [>.............................] - ETA: 36s - loss: 4.6084 - acc
##  12/313 [>.............................] - ETA: 35s - loss: 4.6084 - acc
##  13/313 [>.............................] - ETA: 35s - loss: 4.6079 - acc
##  14/313 [>.............................] - ETA: 35s - loss: 4.6073 - acc
##  15/313 [>.............................] - ETA: 35s - loss: 4.6072 - acc
```

Lab: CNN to the CIFAR data

- Evaluate the model accuracy and loss on the test dataset

```
y_test = to_categorical(y_test, 100)
test_loss, test_acc = model.evaluate(x_test, y_test)
```

```
##
##   1/313 [..............................] - ETA: 21s - loss: 2.5895 - acc
##   4/313 [..............................] - ETA: 6s - loss: 2.6537 - accu
##   7/313 [..............................] - ETA: 6s - loss: 2.7156 - accu
##  10/313 [..............................] - ETA: 6s - loss: 2.7259 - accu
##  13/313 [>.............................] - ETA: 6s - loss: 2.7164 - accu
##  16/313 [>.............................] - ETA: 6s - loss: 2.7825 - accu
##  19/313 [>.............................] - ETA: 6s - loss: 2.8168 - accu
##  22/313 [=>............................] - ETA: 6s - loss: 2.8155 - accu
##  25/313 [=>............................] - ETA: 6s - loss: 2.8001 - accu
##  28/313 [=>............................] - ETA: 6s - loss: 2.8074 - accu
##  31/313 [=>............................] - ETA: 5s - loss: 2.8043 - accu
##  34/313 [==>...........................] - ETA: 5s - loss: 2.8097 - accu
##  37/313 [==>...........................] - ETA: 5s - loss: 2.8068 - accu
##  40/313 [==>...........................] - ETA: 5s - loss: 2.7971 - accu
##  43/313 [===>..........................] - ETA: 5s - loss: 2.8087 - accu
##  45/313 [===>..........................] - ETA: 5s - loss: 2.8117 - accu
```

Lab: CNN to the CIFAR data

```
print('test_loss:', test_loss)
```

```
## test_loss: 2.8095548152923584
```

```
print('test_acc:', test_acc)
```

```
## test_acc: 0.2949000009536743
```

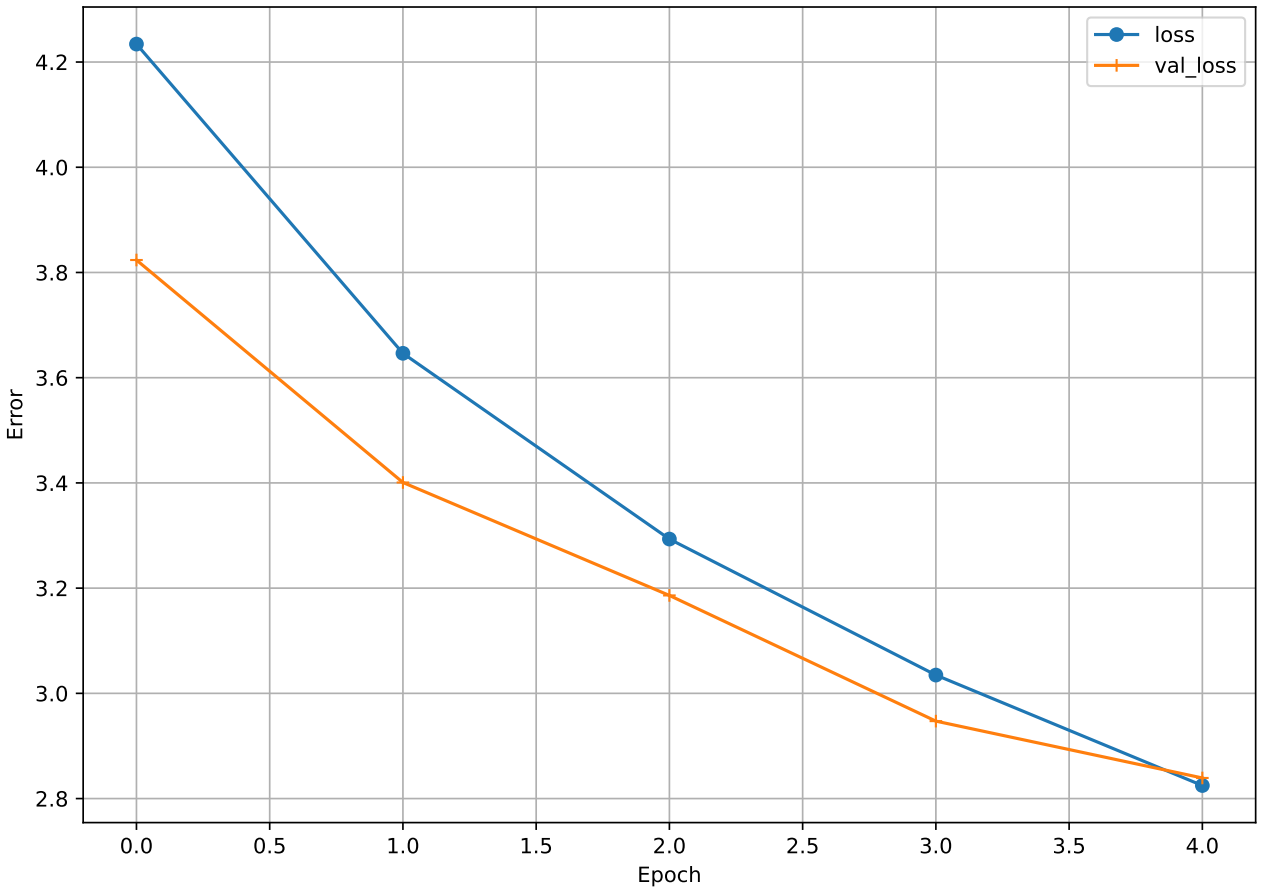
Lab: CNN to the CIFAR data

- Visualizing the fitting procedure using the following function

```
import matplotlib.pyplot as plt
def plot_loss(history):
  plt.plot(history.epoch, history.history['loss'], marker = 'o',
  label='loss')
  plt.plot(history.epoch, history.history['val_loss'], marker = '+',
  label='val_loss')
  plt.xlabel('Epoch')
  plt.ylabel('Error')
  plt.legend()
  plt.grid(True)
```

Lab: CNN to the CIFAR data

```
plot_loss(history)
plt.show()
```



Lab: CNN to the CIFAR data

- Evaluate the model accuracy and loss on the test dataset

```
print('test_acc:', test_acc)
```

```
## test_acc: 0.2949000009536743
```

Lab: Using Pretrained CNN Models

- We now show how to use a CNN pre-trained on the `imagenet` database to classify natural images
- We use 6 images 10_10a.jpg - 10_10f.jpg in Chapter 10 of https://www.statlearning.com/s/Figures-Chapters-7-13.zip
- We first read in the images, and convert them into the array format expected by the `keras` software to match the specifications in `imagenet`. For example, to load the image `10_10a.jpg`

```
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing import image
img=image.load_img('../book_images/10_10a.jpg', color_mode="rgb")
print(type(img))
```

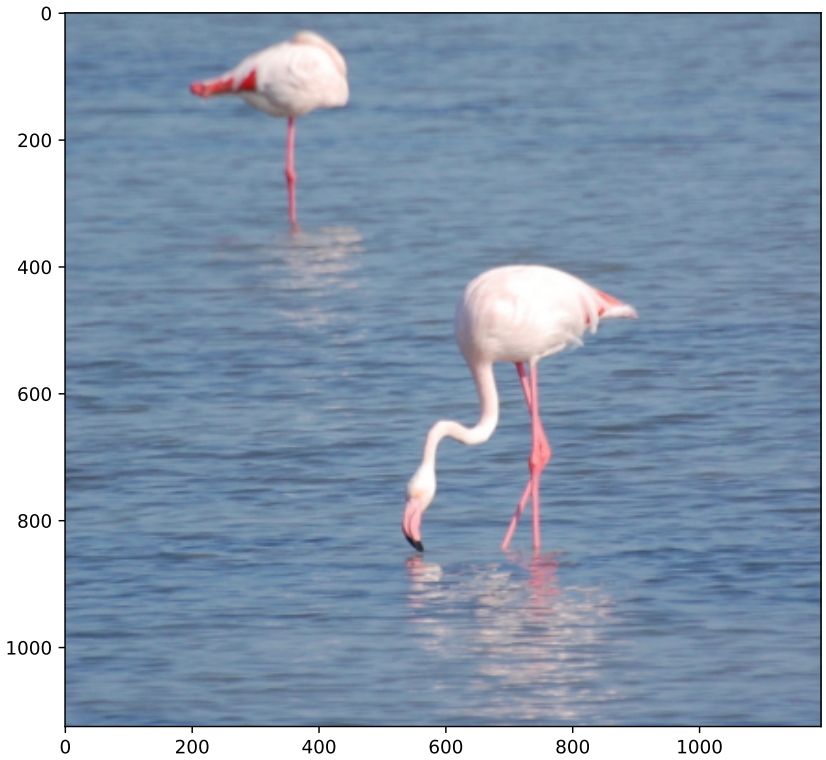
```
## <class 'PIL.JpegImagePlugin.JpegImageFile'>
```

```
array = image.img_to_array(img)
print(array.shape)
```

```
## (1125, 1192, 3)
```

Lab: Using Pretrained CNN Models

```
plt.imshow(img)
plt.show()
```



Lab: Using Pretrained CNN Models

- Image data preprocessing https://keras.io/api/preprocessing/image/
  - We put all 6 images in a folder `book_images`
- The standard folder structure for TF is as what follows if you want to use `tensorflow.keras.preprocessing.image_dataset_from_directory`. We have all images in a single folder.

```
imgdata
|
|___train
|       |___class_1
|       |___class_2
|
|___validation
|       |___class_1
|       |___class_2
|
|___test(optional)
        |___class_1
        |___class_2
```

Lab: Using Pretrained CNN Models

- Generates a `tf.data.Dataset` from image files in the directory.

```
import os
from tensorflow.keras.preprocessing import image_dataset_from_directory
img_dir='../book_images'
image_count = len(os.listdir(img_dir)) #number of images

x=image_dataset_from_directory(img_dir, labels=None, image_size=(224, 224)
```

```
## Found 6 files belonging to 1 classes.
```

```
print(type(x))
```

```
## <class 'tensorflow.python.data.ops.dataset_ops.BatchDataset'>
```

Lab: Using Pretrained CNN Models

- Or use for loop to pre-process the images
  - Preprocesses a tensor or Numpy array encoding a batch of images: https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet50/preprocess_input

```
import os
import numpy as np
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input
img_dir='../book_images'
image_names = os.listdir(img_dir)
image_count = len(os.listdir(img_dir)) #number of images
x=np.zeros((image_count, 224, 224, 3))
for i in range(image_count):
  img_path=img_dir+'/'+image_names[i]
  img = image.load_img(img_path, target_size=(224, 224),color_mode='rgb')
  x[i,:,:,:]=image.img_to_array(img)

x = preprocess_input(x)
print(type(x))
```

Lab: Using Pretrained CNN Models

- We then load the trained network. The model has 50 layers, with a fair bit of complexity.

```
model = tf.keras.applications.ResNet50(weights ="imagenet")
print(model.summary())
```

```
## Model: "resnet50"
## __________________________________________________________________________
##  Layer (type)                   Output Shape         Param #     Connect
## ==========================================================================
##  input_1 (InputLayer)           [(None, 224, 224, 3  0           []
##                                 )]
##
##  conv1_pad (ZeroPadding2D)      (None, 230, 230, 3)  0           ['input
##
##  conv1_conv (Conv2D)            (None, 112, 112, 64  9472        ['conv1
##                                 )
##
##  conv1_bn (BatchNormalization)  (None, 112, 112, 64  256         ['conv1
##                                 )
##
##  conv1_relu (Activation)        (None, 112, 112, 64  0           ['conv1
```

Lab: Using Pretrained CNN Models

- Finally, we classify our six images, and return the top three class choices in terms of predicted probability for each.
- Decodes the prediction of an ImageNet model.
  - https://www.tensorflow.org/api_docs/python/tf/keras/applications/imagenet_utils/decode_predictions

```
pred6=model.predict(x)
```

```
##
## 1/1 [==============================] - ETA: 0s
## 1/1 [==============================] - 3s 3s/step
```

```
type(pred6)
```

```
## <class 'numpy.ndarray'>
```

Lab: Using Pretrained CNN Models

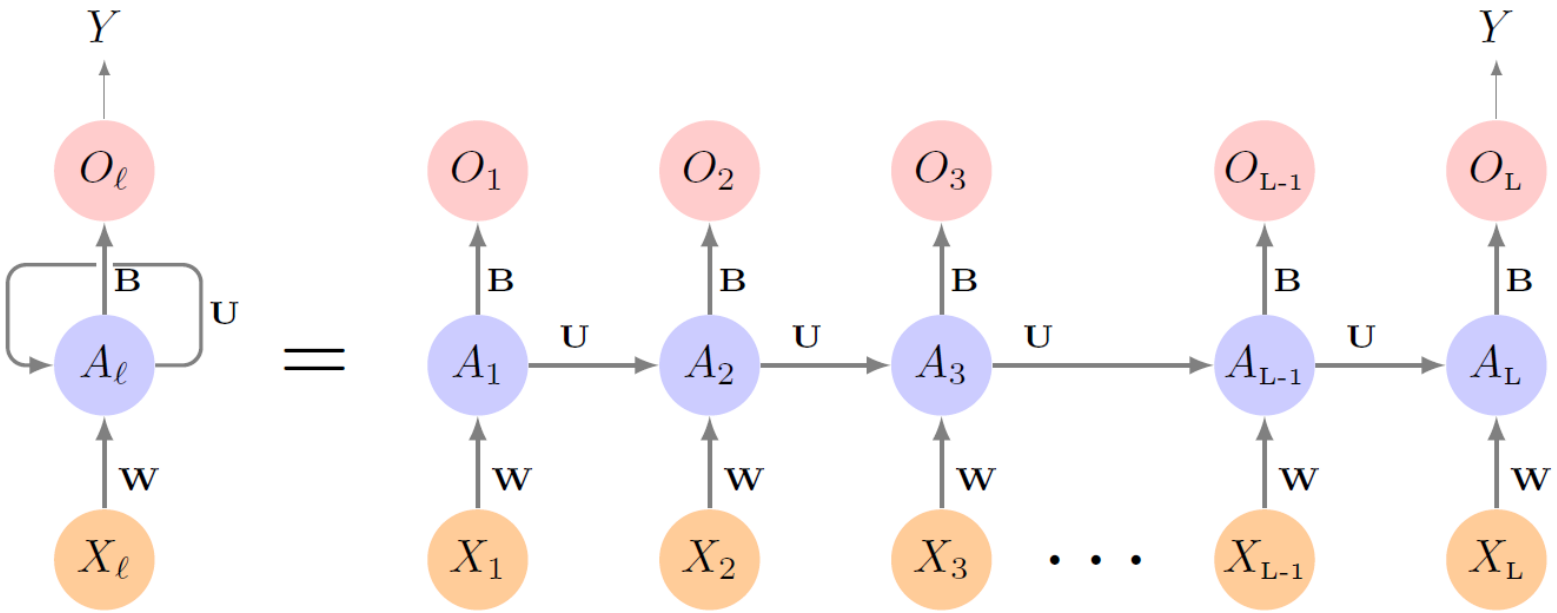
```
from tensorflow.keras.applications.resnet50 import decode_predictions
for i in range(6):
  print('Predicted:', decode_predictions(pred6, top=3)[i])
```

```
## Predicted: [('n01532829', 'house_finch', 0.8807741), ('n01608432', 'kite', 0.044639397), ('n
## Predicted: [('n02097474', 'Tibetan_terrier', 0.4944966), ('n02098413', 'Lhasa', 0.26387113),
## Predicted: [('n02086240', 'Shih-Tzu', 0.38343775), ('n02105641', 'Old_English_sheepdog', 0.0
## Predicted: [('n01742172', 'boa_constrictor', 0.37380132), ('n03532672', 'hook', 0.17335846),
## Predicted: [('n02236044', 'mantis', 0.7781103), ('n06794110', 'street_sign', 0.02557852), ('
## Predicted: [('n02009912', 'American_egret', 0.36105123), ('n02007558', 'flamingo', 0.2940652
```

Recurrent Neural Networks

- Often data arise as sequences:
  - Documents are sequences of words, and their relative positions have meaning.
  - Time-series such as weather data or financial indices.
  - Recorded speech or music.
  - Handwriting, such as doctor's notes.
- RNNs build models that take into account this sequential nature of the data, and build a memory of the past.
- The feature for each observation is a sequence of vectors $X = \{X_1, X_2, \ldots, X_L\}$; each X_l is a vector.
- The target Y is often of the usual kind - e.g. a single variable such as *Sentiment*, or a one-hot vector for multiclass.
- However, Y can also be a sequence, such as the same document in a different language.

Recurrent Neural Networks



- The *hidden layer* is a sequence of vectors A_l, receiving input X_l as well as A_{l-1} (feeding each other). A_l produces an output O_l.
  - A_ls are accumulating/updating memories.
- The *same* weights **W**, **U** and **B** are used at each step in the sequence - hence the term *recurrent*.
- The A_l sequence represents an evolving model for the response that is updated as each element X_l is processed.

Recurrent Neural Networks

Suppose $X_l = (X_{l1}, X_{l2}, \ldots, X_{lp})$ has p components, and $A_l = (A_{l1}, A_{l2}, \ldots, A_{lK})$ has K components. Then the computation at the kth components of hidden unit A_l is (g is an activation function)

$$A_{lk} = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{lj} + \sum_{s=1}^{K} u_{ks} A_{l-1,s}\right)$$

$$O_l = \beta_0 + \sum_{k=1}^{K} \beta_k A_{lk}$$

- Often we are concerned only with the prediction O_L at the last unit. For squared error loss, and n sequence/response pairs, we would minimize

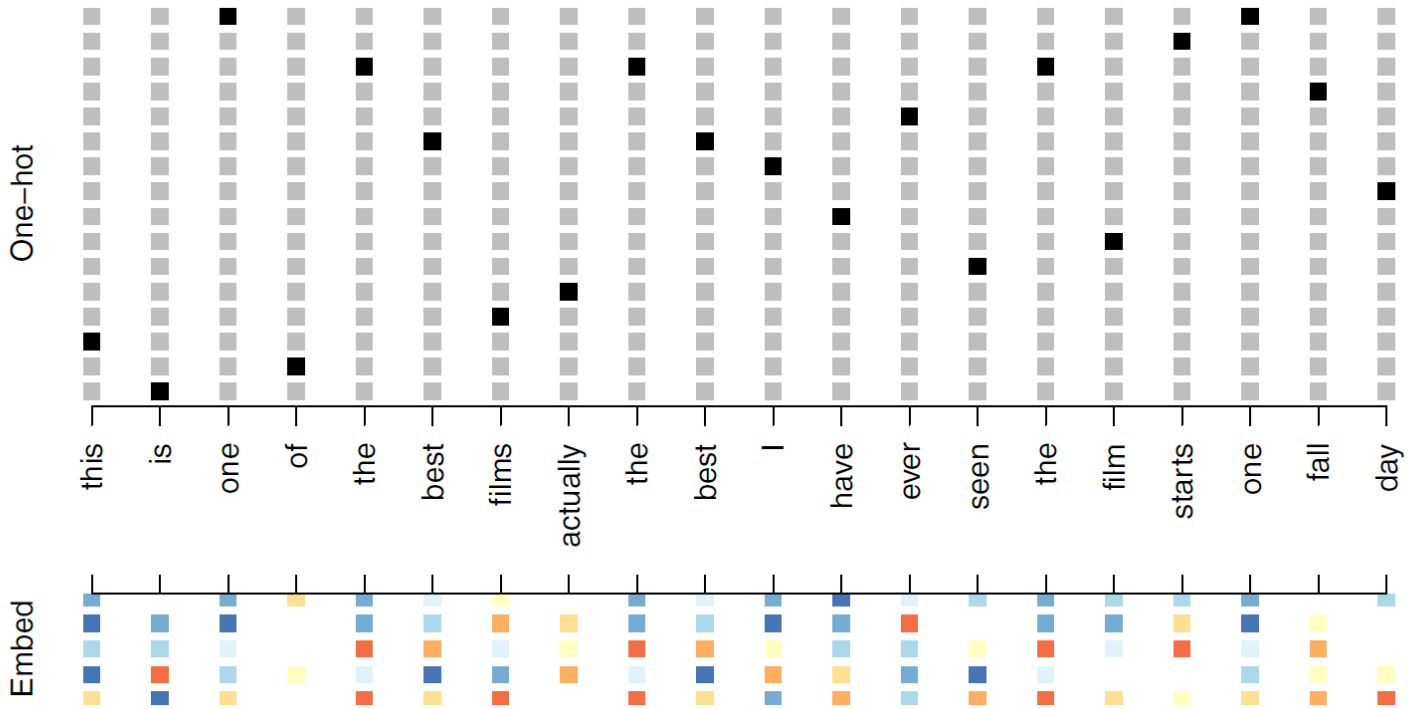
$$\sum_{i=1}^{n} (y_i - o_{iL})^2 = \sum_{i=1}^{n} \left\{ y_i - \left[\beta_0 + \sum_{k=1}^{K} \beta_k \, g\left(\beta_0 + \sum_{j=1}^{p} w_{kj} x_{iLj} + \sum_{s=1}^{K} u_{ks} a_{i,L-1,s} \right) \right] \right\}^2$$

Recurrent Neural Networks

IMDB Reviews:

- The document feature is a sequence of words $\{W_l\}_1^L$. We typically truncate/pad the documents to the same number L of words (we use $L = 500$).
- Each word W_l is represented as a *one-hot encoded* binary vector X_l (dummy variable) of length 10,000, with all zeros and a single one in the position for that word in the dictionary.
- This results in an extremely sparse feature representation, and would not work well.
- Instead we use a lower-dimensional pretrained *word embedding* matrix $\mathbf{E}$ ($m \times 10,000$, next slide).
- This reduces the binary feature vector of length 10,000 to a real feature vector of dimension $m << 10,000$ (e.g. m in the low hundreds.)

Recurrent Neural Networks



- Embeddings are either pretrained on very large corpora of documents or learned during training, using methods similar to principal components: projecting each word in a high-dimensional space to a low-dimensional space.

Embedding

There are various algorithms to obtain word embeddings, such as:

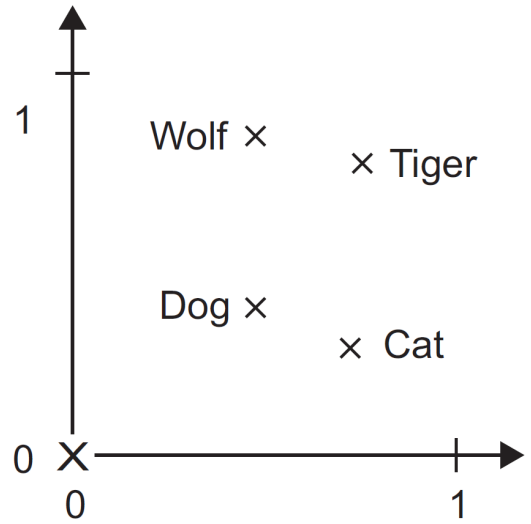
- Continuous Bag-of-Words (CBOW): This method predicts the target word based on the surrounding context words. The model takes in context words as input and outputs a probability distribution over all words in the vocabulary, and the word with the highest probability is selected as the target word.
- *Skip-gram*: This method is the reverse of CBOW, where the model takes the target word as input and predicts the surrounding context words.
- *Word2Vec*: This is a family of algorithms that includes both CBOW and Skip-gram. It was introduced by Google in 2013 and is widely used for various NLP tasks.
- *GloVe (Global Vectors)*: This is a combination of the CBOW and the matrix factorization methods. The model takes in word-context pairs as input and uses matrix factorization to learn the word embeddings.
- *FastText*: This is an extension of the skip-gram model that considers sub-word information in addition to word information. This makes it suitable for handling rare and out-of-vocabulary words.

Embedding

- `N-grams` are overlapping groups of multiple consecutive words or characters.
  - Example: `{"The", "The cat", "cat", "cat sat", "sat", "sat on", "on", "on the", "the", "the mat", "mat"}`
- Collectively, the different units into which you can break down text (words, characters, or n-grams) are called `tokens`, and breaking text into such tokens is called `tokenization`.
- All text-vectorization processes consist of applying some tokenization scheme and then associating numeric vectors with the generated tokens.
- Token embeddings (or word embeddings) are learned from data, they pack more information into far fewer dimensions.
- There are two ways to obtain word embeddings
  - Learn word embeddings jointly with the main task you care about (such as document classification or sentiment prediction). In this setup, you start with random word vectors and then learn word vectors in the same way you learn the weights of a neural network.
  - Load into your model word embeddings that were precomputed using a different machine-learning task than the one you're trying to solve. These are called *pretrained word embeddings*.

Learning word embedding layer

- The geometric relationships between word vectors should reflect the semantic relationships between these words.
- In addition to distance, you may want specific directions in the embedding space to be meaningful.
- In the following figure, four words are embedded on a 2D plane: `cat`, `dog`, `wolf`, and `tiger`.
  - The same vector allows us to go from `cat` to `tiger` and from `dog` to `wolf`: this vector could be interpreted as the 'from pet to wild animal' vector.



Learning word embedding layer

- Common examples of meaningful geometric transformations:
  - `Gender` vector: by adding a `female` vector to the vector `king`, we obtain the vector `queen`.
    - ⋆ By adding a `plural` vector, we obtain `kings`.
- There is no ideal word-embedding space that would perfectly map human language and could be used for any natural-language-processing task.
- It's thus reasonable to **learn** a new embedding space with every new task.

```
from keras.layers import Embedding
embedding_layer = Embedding(samples=1000, sequence_length=64)
```

- The Embedding layer takes at least two arguments
  - the number of possible tokens. Here, 1,000 = maximum word index
  - the dimensionality of the embeddings. Here, 64.
  - each entry is a sequence of integers
- The Embedding layer is best understood as a dictionary that maps integer indices (which stand for specific words) to dense vectors.
- The Embedding layer returns a 3D floating-point tensor of shape (samples, sequence_length, embedding_dimensionality). Such a 3D tensor can then be processed by an RNN layer.

RNN and IMDB Reviews

- We fit a simple *LSTM* (long term and short term memory) RNN for sentiment analysis with the IMDB movie-review data.
- Two tracks of hidden-layer activations are maintained, so that when the activation A_l is computed, it gets input from hidden units both further back in time (long term memory), and closer in time A_{l-1} — a so-called LSTM RNN.
- Data import

```
from keras.datasets import imdb
from keras import preprocessing
from tensorflow.keras.preprocessing import sequence
max_features =10000 #Number of words to consider as features
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
```

RNN and IMDB Reviews

- We first calculate the lengths of the documents.

```
import numpy as np
wc=np.array([ len(x_train[i]) for i in range(len(x_train))])
np.mean(wc)
```

```
## 238.71364
```

```
np.median(wc)
```

```
## 178.0
```

- We see that over 91% of the documents have fewer than 500 words. Our RNN requires all the document sequences to have the same length. We hence restrict the document lengths to the last $L = 500$ words, and pad the beginning of the shorter ones with blanks.

```
sum(wc<=500)/len(wc)
```

```
## 0.91568
```

RNN and IMDB Reviews

```
max_len = 500
x_train = sequence.pad_sequences(x_train, maxlen=max_len)
x_test = sequence.pad_sequences(x_test, maxlen=max_len)
print(x_train.shape)
```

```
## (25000, 500)
```

```
print(x_test.shape)
```

```
## (25000, 500)
```

- The following shows the last few words in the first document.

```
x_train[0, 489:500]
```

```
## array([  16, 4472,  113,  103,   32,   15,   16, 5345,   19,  178,   32])
```

- At this stage, each of the 500 words in the document is represented using an integer corresponding to the location of that word in the 10,000-word dictionary.

RNN and IMDB Reviews

- The first layer of the RNN is an embedding layer of size 32, which will be learned during training. This layer one-hot encodes each document as a matrix of dimension $500 times 10,000$, and then maps these $10,000$ dimensions down to 32.
- The second layer is an LSTM with 32 units, and the output layer is a single sigmoid for the binary classification task.

```
import tensorflow as tf
model=tf.keras.Sequential()#Define the model
model.add(tf.keras.layers.Embedding(10000, 32))
model.add(tf.keras.layers.LSTM(32, dropout=0.2, recurrent_dropout=0.2))
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
```

RNN and IMDB Reviews

- The rest is now similar to other networks we have fit. Compile the model first

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
print(model.summary())
```

```
## Model: "sequential_5"
## _________________________________________________________________
##  Layer (type)                 Output Shape              Param #
## =================================================================
##  embedding (Embedding)        (None, None, 32)          320000
##
##  lstm (LSTM)                  (None, 32)                8320
##
##  dense_11 (Dense)             (None, 1)                 33
##
## =================================================================
## Total params: 328,353
## Trainable params: 328,353
## Non-trainable params: 0
## _________________________________________________________________
## None
```

RNN and IMDB Reviews

- Model fit using epochs=10 which will take very long time, about 5 minutes/epoch on my computer.

```
history=model.fit(x_train, y_train, batch_size=32,
epochs=10,validation_data=(x_test, y_test))
```

RNN and IMDB Reviews

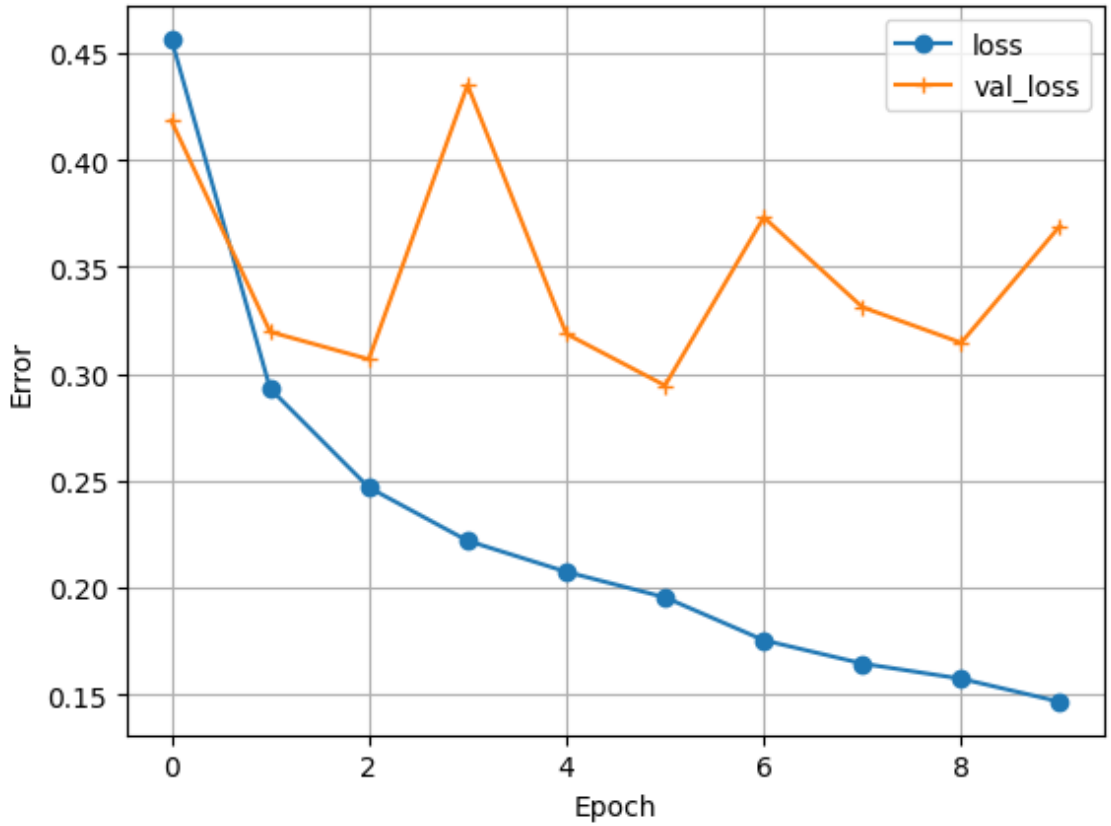
- Visualizing the fitting procedure using the function we defined before

```
import matplotlib.pyplot as plt
def plot_loss(history):
  plt.plot(history.epoch, history.history['loss'], marker = 'o',
  label='loss')
  plt.plot(history.epoch, history.history['val_loss'], marker = '+',
  label='val_loss')
  plt.xlabel('Epoch')
  plt.ylabel('Error')
  plt.legend()
  plt.grid(True)
```

RNN and IMDB Reviews

```
plot_loss(history)
plt.show()
```

- Here is what I got. It can be seen that the model overfit quickly. So I re-fit the model using epochs=6.




```
history=model.fit(x_train, y_train, batch_size=32,
epochs=6,validation_data=(x_test, y_test))
```

RNN with pretrained word embeddings

- Instead of learning word embeddings jointly with the problem we want to solve, we can load embedding vectors from a precomputed embedding space.
- `Word2vec` algorithm (https://code.google.com/ archive/p/word2vec)
  - developed by Tomas Mikolov at Google in 2013.
  - `Word2vec` dimensions capture specific semantic properties, such as gender.
- `Global Vectors`(GloVe, https://nlp.stanford .edu/projects/glove)
  - developed by Stanford researchers in 2014.
  - This embedding technique is based on factorizing a matrix of word co-occurrence statistics.
- We'll start from scratch by downloading the original text data.

RNN with pretrained word embeddings

- **Step 1**: DOWNLOADING THE IMDB DATA AS RAW TEXT
  - First, head to http://mng.bz/0tIo and download the raw IMDB dataset. Uncompress it.
  - `os.path`: https://docs.python.org/3/library/os.path.html
  - `os.listdir`: https://docs.python.org/3/library/os.html#os.listdir

```
import os
imdb_dir = '../aclImdb'
train_dir = os.path.join(imdb_dir, 'train')
labels = [] #review labels
texts = []  #review texts

for label_type in ['neg', 'pos']:
  dir_name = os.path.join(train_dir, label_type)
  for fname in os.listdir(dir_name):
    if fname[-4:] == '.txt':
      f = open(os.path.join(dir_name, fname)) #path to the .txt file
      texts.append(f.read()) #read the .txt file f
      f.close() #close the file
      if label_type == 'neg':
        labels.append(0)
      else:
        labels.append(1)
```

RNN with pretrained word embeddings

- **Step 2**: TOKENIZING THE DATA. Let's vectorize the text and prepare a training and validation split
  - Assume that little training data is available. So we restrict the training data to the first 200 samples

```
from keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

maxlen = 100 #Cuts off reviews after 100 words
training_samples = 200 #Trains on 200 samples
validation_samples = 10000 #Validates on 10,000 samples
max_words = 10000 #Considers only the top 10,000 words in the dataset

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

```
## Found 88582 unique tokens.
```

```
data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
print('Shape of data tensor:', data.shape)
```

```
## Shape of data tensor: (25000, 100)
```

RNN with pretrained word embeddings

- **Step 2**: TOKENIZING THE DATA. Let's vectorize the text and prepare a training and validation split
  - Assume that little training data is available. So we restrict the training data to the first 200 samples

```
#Splits the data into a training set and a validation set,
#but first shuffles the data, because you're starting with data
#in which samples are ordered (all negative first, then all positive)
indices = np.arange(data.shape[0])
np.random.shuffle(indices)

data = data[indices]
labels = labels[indices]
#the first 200 rows are training set
x_train = data[:training_samples]
y_train = labels[:training_samples]

#valaidata samples
x_val = data[training_samples:training_samples + validation_samples]
y_val = labels[training_samples: training_samples + validation_samples]
```

RNN with pretrained word embeddings

- **Step 3**: DOWNLOADING THE GLOVE WORD EMBEDDINGS - the precomputed embeddings from 2014 English Wikipedia
  - https://nlp.stanford.edu/projects/glove
  - It's an 822 MB zip file called glove.6B.zip, containing 100-dimensional embedding vectors for 400,000 words (or nonwordtokens). Unzip it.

RNN with pretrained word embeddings

- **Step 4**: PREPROCESSING/PARSING THE EMBEDDINGS.
- Let's parse the unzipped file (a .txt file) to build an index that maps words (as strings) to their vector representation (as number vectors).
- Open a file and return a corresponding file object: https://docs.python.org/3/library/functions.html#open
- File input and output: https://docs.python.org/3/tutorial/inputoutput.html

```
glove_dir = '../glove.6B'
embeddings_index = {}  #map words to numerical vectors
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))

for line in f: #loop over each line
  values = line.split() #splitting the line into separate values, using whi
  word = values[0] #stores the word as the key
  coefs = np.asarray(values[1:], dtype='float32') #value of the dic
  embeddings_index[word] = coefs

f.close()

print('Found %s word vectors.' % len(embeddings_index))
#list(embeddings_index)[:10]
```

RNN with pretrained word embeddings

- **Step 4**: PREPROCESSING/PARSING THE EMBEDDINGS.
- Next, we'll build an embedding matrix that we can load into an Embedding layer
  - It must be a matrix of shape (`max_words`, `embedding_dim`), where each entry `i` contains the **`embedding_dim`-dimensional vector** for the word of index `i` in the reference word index (built during tokenization).
  - Note that index 0 isn't supposed to stand for any word or token—it's a placeholder.

```
embedding_dim = 100

embedding_matrix = np.zeros((max_words, embedding_dim))
#embedding_matrix.shape
for word, i in word_index.items():
  if i < max_words:
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
      embedding_matrix[i] = embedding_vector
      #Words not found in the embedding index will be all zeros.
```

RNN with pretrained word embeddings

- **Step 5**: DEFINING A MODEL
  - We'll use the same model architecture as before.

```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words,embedding_dim,input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

RNN with pretrained word embeddings

- **Step 5**: DEFINING A MODEL
  - We'll use the same model architecture as before.

```
model.summary()
```

```
## Model: "sequential_6"
## _________________________________________________________________
##  Layer (type)                 Output Shape              Param #
## =================================================================
##  embedding_1 (Embedding)      (None, 100, 100)          1000000
##
##  flatten_1 (Flatten)          (None, 10000)             0
##
##  dense_12 (Dense)             (None, 32)                320032
##
##  dense_13 (Dense)             (None, 1)                 33
##
## =================================================================
## Total params: 1,320,065
## Trainable params: 1,320,065
## Non-trainable params: 0
## _________________________________________________________________
```

RNN with pretrained word embeddings

- **Step 6**: LOADING THE GLOVE EMBEDDINGS IN THE MODEL
  - The Embedding layer has a single weight matrix: a 2D float matrix where each entry `i` is the word vector meant to be associated with index `i`.
  - Load the `GloVe` matrix we prepared into the `Embedding` layer, the first layer in the model.
    - ★ Additionally, we'll freeze the `Embedding` layer since the pretrained parts shouldn't be updated during training, to avoid forgetting what they already know.

```
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
```

RNN with pretrained word embeddings

- **Step 7**: COMPILE AND TRAIN THE MODEL

```
model.compile(optimizer='rmsprop',
loss='binary_crossentropy',metrics=['acc'])
history = model.fit(x_train, y_train,
epochs=15, batch_size=32,
validation_data=(x_val, y_val))
```

```
## Epoch 1/15
##
## 1/7 [===>..........................] - ETA: 3s - loss: 0.8502 - acc: 0.2
## 7/7 [==============================] - 1s 157ms/step - loss: 1.7146 - ac
## Epoch 2/15
##
## 1/7 [===>..........................] - ETA: 0s - loss: 0.6508 - acc: 0.6
## 7/7 [==============================] - 1s 126ms/step - loss: 0.6380 - ac
## Epoch 3/15
##
## 1/7 [===>..........................] - ETA: 0s - loss: 0.5804 - acc: 0.7
## 7/7 [==============================] - 1s 125ms/step - loss: 0.5594 - ac
## Epoch 4/15
##
```

RNN with pretrained word embeddings

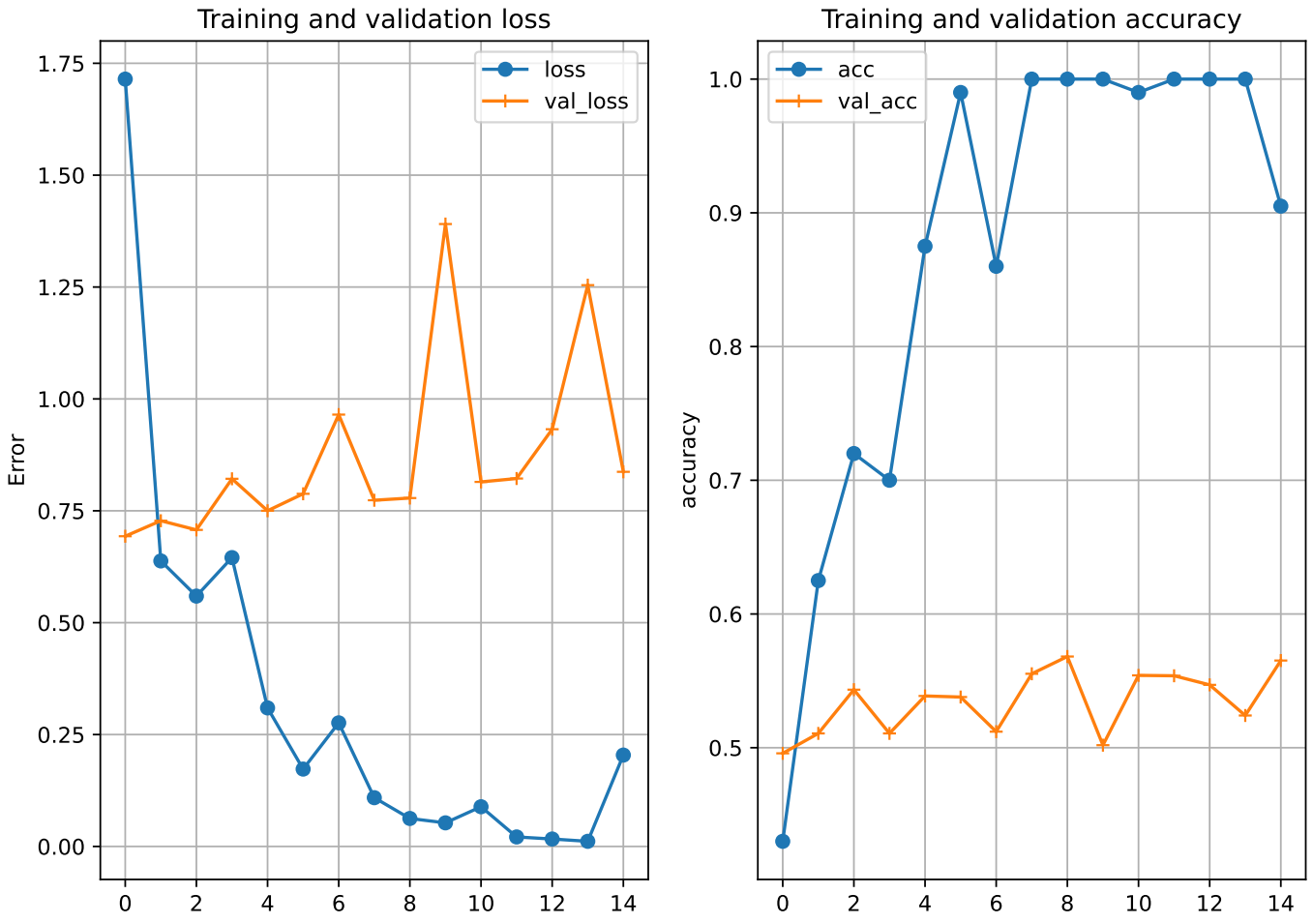
- **Step 8**: plot the model's performance over time

```
import matplotlib.pyplot as plt

def plot_loss_acc(history):
  fig, axs = plt.subplots(1, 2)
  axs[0].plot(history.epoch, history.history['loss'],
  marker = 'o', label='loss')
  axs[0].plot(history.epoch, history.history['val_loss'],
  marker = '+', label='val_loss')
  axs[0].set(xlabel='Epoch', ylabel='Error',
       title='Training and validation loss')
  axs[0].legend()
  axs[0].grid(True)

  axs[1].plot(history.epoch, history.history['acc'],
  marker = 'o', label='acc')
  axs[1].plot(history.epoch, history.history['val_acc'],
  marker = '+', label='val_acc')
  axs[1].set(xlabel='Epoch', ylabel='accuracy',
       title='Training and validation accuracy')
  axs[1].legend()
  axs[1].grid(True)
```

RNN with pretrained word embeddings

- **Step 8**: plot the model's performance over time

```
plot_loss_acc(history)
plt.show()
```



RNN with pretrained word embeddings

- The model quickly starts overfitting, which is unsurprising given the small number of training samples.
- Validation accuracy has high variance for the same reason, but it seems to reach the high 50s.
- We can also train the same model without loading the pretrained word embeddings and without freezing the embedding layer.
  - In that case, you'll learn a taskspecific embedding of the input tokens, which is generally more powerful than pretrained word embeddings **if lots of data is available**

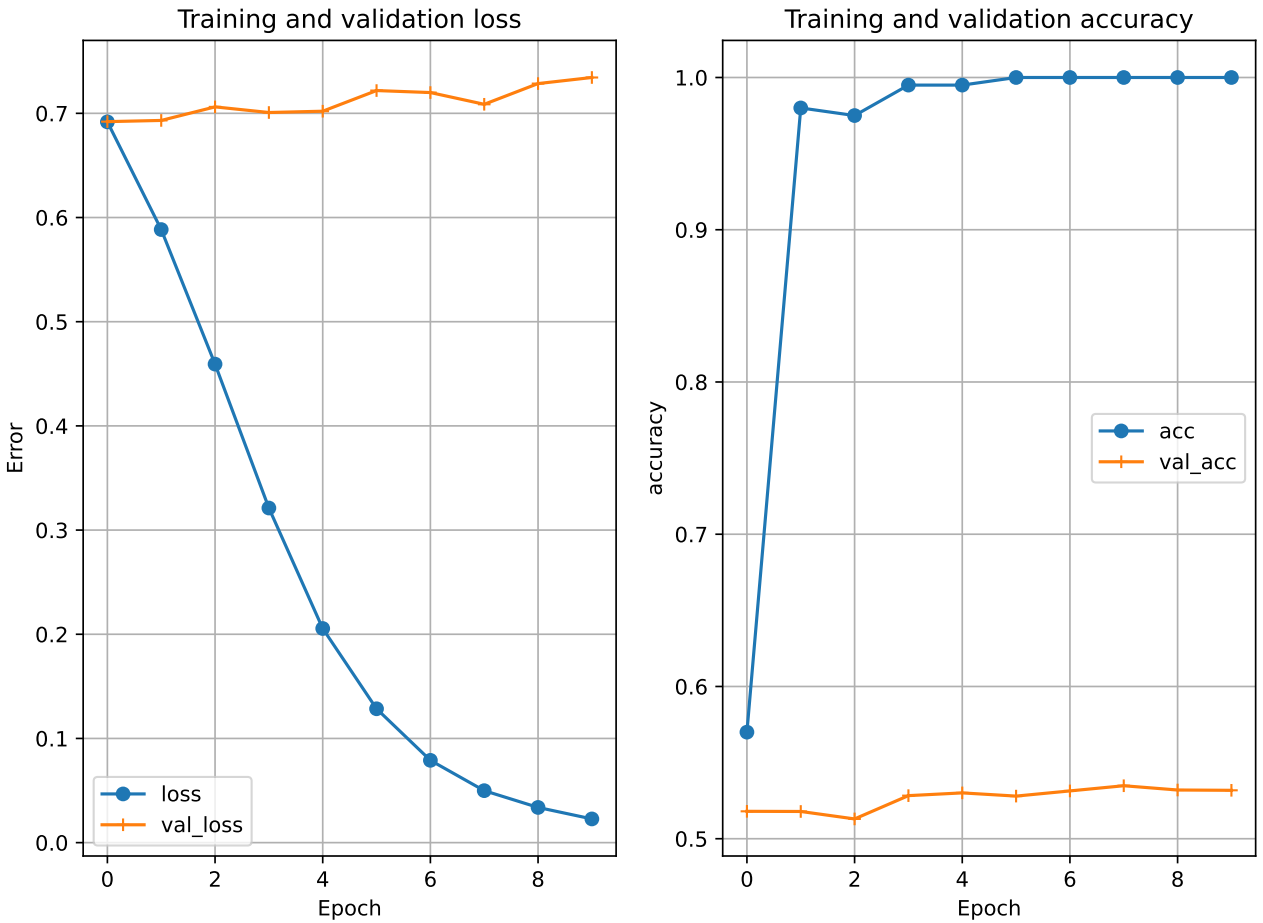
```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model2 = Sequential()
model2.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model2.add(Flatten())
model2.add(Dense(32, activation='relu'))
model2.add(Dense(1, activation='sigmoid'))
#model2.summary()

model2.compile(optimizer='rmsprop',loss='binary_crossentropy',metrics=['acc'])
history2 = model2.fit(x_train, y_train,epochs=10,
batch_size=32,validation_data=(x_val, y_val))
```

```
## Epoch 1/10
##
```

RNN with pretrained word embeddings

- Pretrained word embeddings outperform jointly learned embeddings:

```
plot_loss_acc(history2)
plt.show()
```



RNN with pretrained word embeddings

- **Step 9**: EVALUATE THE MODEL ON THE TEST DATA
- First, we need to tokenize the test data.

```
import os
imdb_dir = '../aclImdb'
test_dir = os.path.join(imdb_dir, 'test')
labels = []
texts = []

for label_type in ['neg', 'pos']:
  dir_name = os.path.join(test_dir, label_type)
  for fname in sorted(os.listdir(dir_name)):
    if fname[-4:] == '.txt':
      f = open(os.path.join(dir_name, fname))
      texts.append(f.read())
      f.close()
      if label_type == 'neg':
        labels.append(0)
      else:
        labels.append(1)

sequences = tokenizer.texts_to_sequences(texts)
x_test = pad_sequences(sequences, maxlen=maxlen)
y_test = np.asarray(labels)
```

RNN with pretrained word embeddings

- **Step 9**: EVALUATE THE MODEL ON THE TEST DATA
  - Next, load and evaluate the first model.

```
model.load_weights('pre_trained_glove_model.h5')
test_loss, test_acc=model.evaluate(x_test, y_test)
```

```
##
##   1/782 [..............................] - ETA: 12s - loss: 0.9891 - acc
##  22/782 [..............................] - ETA: 2s - loss: 0.9414 - acc:
##  47/782 [>.............................] - ETA: 2s - loss: 0.9747 - acc:
##  67/782 [=>............................] - ETA: 1s - loss: 0.9920 - acc:
##  90/782 [==>...........................] - ETA: 1s - loss: 0.9904 - acc:
## 113/782 [===>..........................] - ETA: 1s - loss: 1.0068 - acc:
## 135/782 [====>.........................] - ETA: 1s - loss: 1.0089 - acc:
## 159/782 [=====>........................] - ETA: 1s - loss: 1.0142 - acc:
## 183/782 [======>.......................] - ETA: 1s - loss: 1.0135 - acc:
## 207/782 [======>.......................] - ETA: 1s - loss: 1.0158 - acc:
## 232/782 [=======>......................] - ETA: 1s - loss: 1.0221 - acc:
## 256/782 [========>.....................] - ETA: 1s - loss: 1.0215 - acc:
## 277/782 [=========>....................] - ETA: 1s - loss: 1.0197 - acc:
## 299/782 [==========>...................] - ETA: 1s - loss: 1.0227 - acc:
## 326/782 [===========>..................] - ETA: 1s - loss: 1.0248 - acc:
```

RNN with pretrained word embeddings

- **Step 9**: EVALUATE THE MODEL ON THE TEST DATA
  - We get an appalling test accuracy around 50%. Working with just a handful of training samples is difficult!

```
print('test_loss:', test_loss)
```

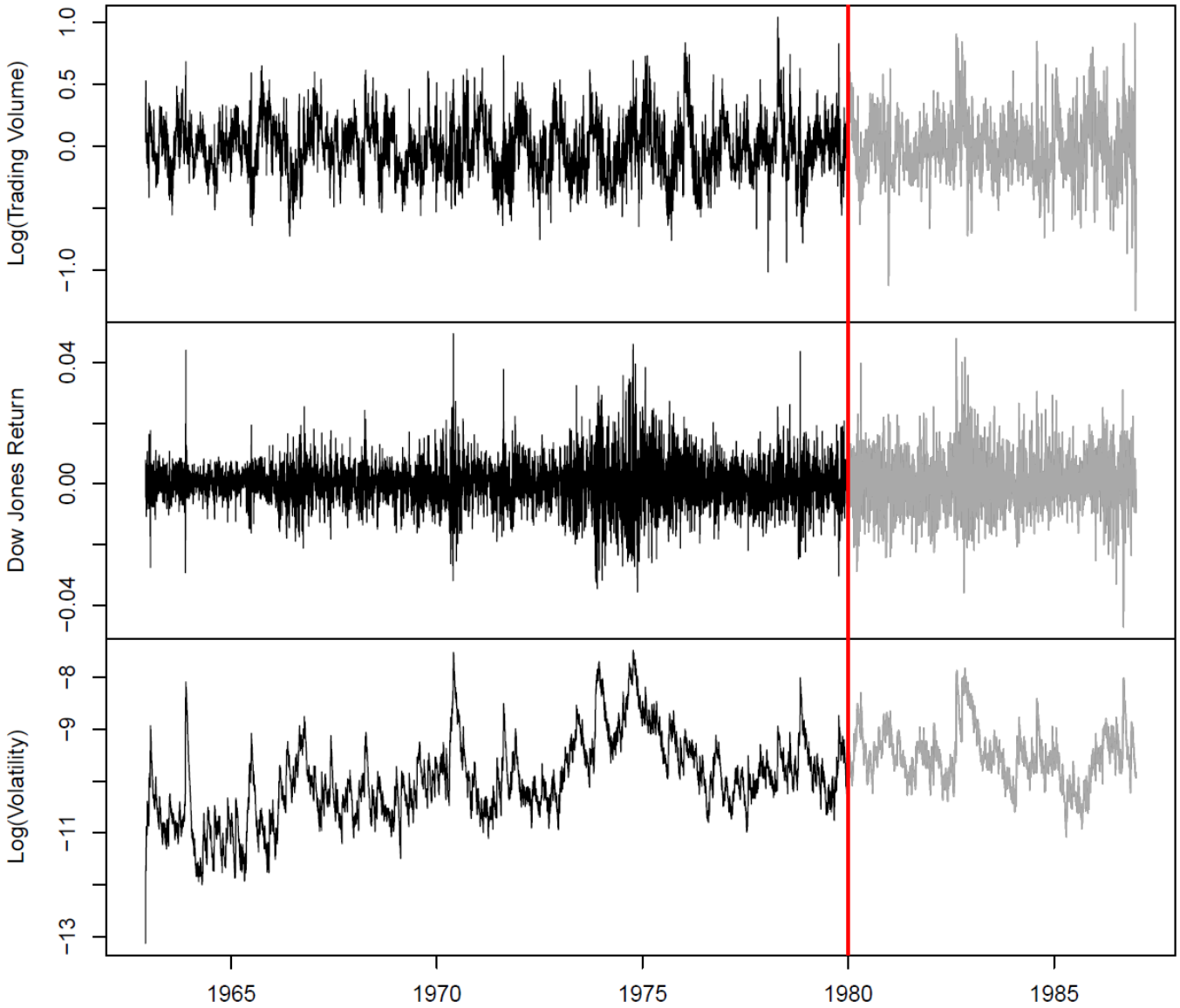
```
## test_loss: 0.857824981212616
```

```
print('test_acc:', test_acc)
```

```
## test_acc: 0.557640016078949
```

RNN for time-series data

- Time Series Forecasting



RNN for time-series data

Shown in previous slide are three daily time series for the period December 3, 1962 to December 31, 1986 (6,051 trading days):

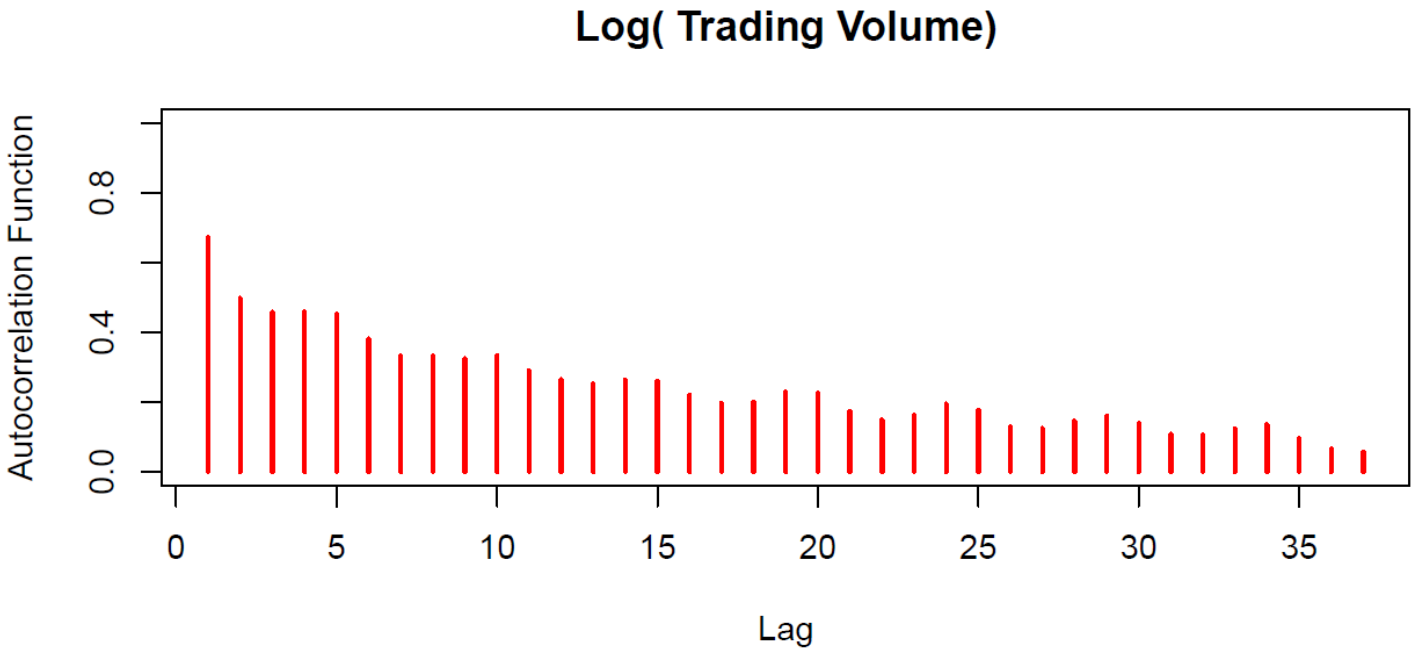
- `Log trading volume`. This is the fraction of all outstanding shares that are traded on that day, relative to a 100-day moving average of past turnover, on the log scale.
- `Dow Jones return`. This is the difference between the log of the Dow Jones Industrial Index on consecutive tradingdays.
- Log volatility. This is based on the absolute values of daily price movements.

Goal: predict Log trading volume tomorrow, given its observed values up to today, as well as those of Dow Jones return and Log volatility.

These data were assembled by LeBaron and Weigend (1998), `IEEE Transactions on Neural Networks`, 9(1): 213-220.

RNN for time-series data

- Autocorrelation: The autocorrelation at lag l is the correlation of all pairs (v_t, v_{t-l}) that are l trading days apart.
- These sizable correlations give us confidence that past values will be helpful in predicting the future.



- This is a curious prediction problem: the response v_t is also a feature v_{t-l}!

RNN for time-series data

RNN Forecaster:

- We extract many short mini-series of input sequences $X = \{X_1, X_2, \ldots, X_L\}$ with a predefined length L known as the `lag`:

$$X_1 = \begin{pmatrix} v_{t-L} \\ r_{t-L} \\ z_{t-L} \end{pmatrix}, X_2 = \begin{pmatrix} v_{t-L+1} \\ r_{t-L+1} \\ z_{t-L+1} \end{pmatrix}, \cdots, X_L = \begin{pmatrix} v_{t-1} \\ r_{t-1} \\ z_{t-1} \end{pmatrix}, \text{ and } Y = v_t.$$

- Since $T = 6,051$, with $L = 5$ we can create $6,046$ such (X, Y) pairs.
- We use the frist $4,281$ as training data, and the following $1,770$ as test data. We fit an RNN with 12 hidden units per lag step (i.e. per A_l.)

RNN for time-series data

Autoregression Forecaster:

- The RNN forecaster is similar in structure to a traditional `autoregression` procedure.

$$\mathbf{y} = \begin{bmatrix} v_{L+1} \\ v_{L+2} \\ v_{L+3} \\ \vdots \\ v_T \end{bmatrix}, \mathbf{M} = \begin{bmatrix} 1 & v_L & v_{L-1} & \cdots & v_2 & v_1 \\ 1 & v_{L+1} & v_L & \cdots & v_3 & v_2 \\ 1 & v_{L+2} & v_{L+1} & \cdots & v_4 & v_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & v_{T-1} & v_{T-2} & \cdots & v_{T-L+1} & v_{T-L} \end{bmatrix}$$

- Fit an OLS regression of $\mathbf{y}$ on $\mathbf{M}$, giving

$$\hat{v}_t = \hat{\beta}_0 + \hat{\beta}_1 v_{t-1} + \hat{\beta}_2 v_{t-2} + \cdots + \hat{\beta}_L v_{t-L}$$

  Known as an 1 or AR(L).

RNN for time-series data

- We now show how to fit a RNN model for time series prediction
- Import and normalize the data

```
# the data NYSE was downloaded from R package `ISLR2`
import pandas as pd
import tensorflow as tf
from sklearn.preprocessing import StandardScaler
import numpy as np
NYSE=pd.read_csv('../data/NYSE.csv')
NYSE.columns
```

```
## Index(['date', 'day_of_week', 'DJ_return', 'log_volume', 'log_volatility',
##        'train'],
##       dtype='object')
```

```
xdata=NYSE[['DJ_return','log_volume', 'log_volatility']]
istrain =NYSE["train"] #6051 rows

#normalizer = tf.keras.layers.Normalization(axis=-1)
#normalizer.adapt(np.array(xdata))
#xdata=normalizer(xdata)
xdata = StandardScaler().fit_transform(xdata)

xdata=pd.DataFrame({'DJ_return':xdata[:,0],'log_volume':xdata[:,1],
'log_volatility':xdata[:,2]})
```

RNN for time-series data

- We first write functions to create lagged versions of the three time series. We start with a function that takes as input a data matrix and a lag L, and returns a lagged version of the matrix. It simply inserts L rows of NA at the top, and truncates the bottom.

```
# lagged version for all three variables
import numpy as np
def lagm(x, k): #x is a data frame; k=1,2,3,4,5
  n=x.shape[0]  #number of rows
  m=x.shape[1]  #number of columns
  pad = np.full((k, m), np.nan)
  return(np.vstack((pad, x.iloc[0:(n-k), :])))
  # an array of 3 columns
```

RNN for time-series data

- We now use this function to create a data frame with all the required lags, as well as the response variable.

```
arframe = pd.DataFrame({'log_volume':xdata["log_volume"]})
for i in range(1,6):  #i=1,2,3,4,5
  column_name1 = 'L'+str(i)+'DJ_return'
  column_name2 = 'L'+str(i)+'log_volume'
  column_name3 = 'L'+str(i)+'log_volatility'
  new_columns = {column_name1: lagm(xdata, i)[:,0],
  column_name2: lagm(xdata, i)[:,1],
  column_name3: lagm(xdata, i)[:,2]}
  arframe = arframe.assign(**new_columns)

arframe.columns
```

```
## Index(['log_volume', 'L1DJ_return', 'L1log_volume', 'L1log_volatility',
##        'L2DJ_return', 'L2log_volume', 'L2log_volatility', 'L3DJ_return',
##        'L3log_volume', 'L3log_volatility', 'L4DJ_return', 'L4log_volume',
##        'L4log_volatility', 'L5DJ_return', 'L5log_volume', 'L5log_volatil
##       dtype='object')
```

RNN for time-series data

```
arframe.head(5)
```

```
##    log_volume  L1DJ_return  ...  L5log_volume  L5log_volatility
## 0    0.175075          NaN  ...           NaN               NaN
## 1    1.517291    -0.549823  ...           NaN               NaN
## 2    2.283789     0.905200  ...           NaN               NaN
## 3    0.935176     0.434813  ...           NaN               NaN
## 4    0.224779    -0.431397  ...           NaN               NaN
##
## [5 rows x 16 columns]
```

- If we look at the first five rows of this frame, we will see some missing values in the lagged variables (due to the construction above). We remove these rows, and adjust istrain accordingly.

```
arframe = arframe.iloc[5:]
istrain = istrain.iloc[5:]
```

RNN for time-series data

- We now fit the linear AR model to the training data, and predict on the test data.

```
import statsmodels.api as sm
y_train = arframe[istrain].log_volume
X_train = arframe[istrain].iloc[:,1:]
y_test = arframe[~istrain].log_volume
X_test = arframe[~istrain].iloc[:,1:]

lm_fit=sm.OLS(y_train, X_train).fit()
arpred= lm_fit.predict(X_test) #predicted values

np.mean((arpred-y_test)**2) #prediction error
```

```
## 0.6184775861226861
```

```
np.var(y_test)
```

```
## 1.0536865829448807
```

```
1-np.mean((arpred-y_test)**2)/np.var(y_test) #R-square
```

```
## 0.41303458150321803
```

RNN for time-series data

- We refit this model, including the factor variable day_of_week.
  - To avoid creating dummy variables for the categorical variable, you can use `sm.formula.ols`; but `'formula=y~.'` is not supported.

```
arframed = arframe.assign(day=NYSE.loc[5:,"day_of_week"])
arframed['day'] = arframed['day'].astype('category')
```

- Let's create dummy variables for the categorical variable

```
dummies = pd.get_dummies(NYSE.loc[5:,"day_of_week"])
arframed = arframe.assign(**dummies)
X_traind = arframed[istrain].iloc[:,1:]
X_testd = arframed[~istrain].iloc[:,1:]
```

RNN for time-series data

```
arfitd =sm.OLS(y_train, X_traind).fit()
arpredd = arfitd.predict(X_testd) #predicted values

np.mean((arpredd-y_test)**2) #prediction error decreased
```

```
## 0.5694582616074634
```

```
np.var(y_test)
```

```
## 1.0536865829448807
```

```
1-np.mean((arpredd-y_test)**2)/np.var(y_test) #R-square
```

```
## 0.45955631321040347
```

RNN for time-series data

- To fit the RNN, we need to reshape these data, since it expects a sequence of L=5 feature vectors $X = \{X_l\}_1^L$ for each observation. These are lagged versions of the time series going back L time points.
- Note that index $L1$ is furthest back in time, and index $L5$ is the closest. So we need to reverses the order of lagged variables

```
n=arframe.shape[0]
xrnn=arframe.iloc[:, 1:] #dim=n*15
# reverses the order of lagged variables:
xrnn=xrnn.iloc[:, ::-1]
print(xrnn.columns)
```

```
## Index(['L5log_volatility', 'L5log_volume', 'L5DJ_return', 'L4log_volatil
##        'L4log_volume', 'L4DJ_return', 'L3log_volatility', 'L3log_volume'
##        'L3DJ_return', 'L2log_volatility', 'L2log_volume', 'L2DJ_return',
##        'L1log_volatility', 'L1log_volume', 'L1DJ_return'],
##       dtype='object')
```

```
xrnn.iloc[:,:3].head()
```

```
##    L5log_volatility  L5log_volume  L5DJ_return
## 5         -4.357078      0.175075    -0.549823
## 6         -2.529058      1.517291     0.905200
```

RNN for time-series data

```
xrnn =np.array(xrnn).reshape(n,5,3)
xrnn[0,:,:]

# Final step: rearrange the coordinates of the array (like a partial transp
#into the format that the RNN module in keras expects.
# swap the second and third dimensions
# each element becomes a 3*5 matrix:
#xrnn = np.transpose(xrnn, (0, 2, 1))
#xrnn[0,:,:]
```

```
## array([[-4.35707786,  0.17507497, -0.54982334],
##        [-2.52905765,  1.51729071,  0.90519995],
##        [-2.41803694,  2.28378937,  0.43481275],
##        [-2.36652094,  0.93517558, -0.43139673],
##        [-2.50097011,  0.22477858,  0.04634026]])
```

RNN for time-series data

- Now we are ready to proceed with the RNN, which uses 12 hidden units.
  - We use a simple RNN, the output from the previous time step is fed back into the network as an input to the current time step.
- We specify two forms of dropout for the units feeding into the hidden layer.
  - The first is for the input sequence feeding into this layer,
  - and the second is for the previous hidden units feeding into the layer.

```
from keras.models import Sequential
from keras.layers import SimpleRNN, Dense

# define the model
model = Sequential()
model.add(SimpleRNN(units=12, input_shape=(5, 3),
dropout=0.1, recurrent_dropout=0.1))
model.add(Dense(units=1))

# compile the model
model.compile(optimizer='rmsprop', loss='mse', metrics=['acc'])
```

RNN for time-series data

- We fit the model in a similar fashion to previous networks.

```
history = model.fit(xrnn[istrain,:,:], arframe[istrain][ "log_volume"],
batch_size=64, epochs=75,
validation_data=(xrnn[~istrain,:,:], arframe[~istrain][ "log_volume"]))
```

```
## Epoch 1/75
##
##  1/67 [..............................] - ETA: 47s - loss: 2.0958 - acc:
## 19/67 [=======>......................] - ETA: 0s - loss: 1.5090 - acc: 0
## 32/67 [============>.................] - ETA: 0s - loss: 1.4083 - acc: 0
## 47/67 [====================>.........] - ETA: 0s - loss: 1.3078 - acc: 0
## 59/67 [=========================>....] - ETA: 0s - loss: 1.2557 - acc: 0
## 67/67 [==============================] - 1s 7ms/step - loss: 1.2031 - ac
## Epoch 2/75
##
##  1/67 [..............................] - ETA: 0s - loss: 0.7319 - acc: 0
## 18/67 [=======>......................] - ETA: 0s - loss: 0.8050 - acc: 0
## 37/67 [==============>...............] - ETA: 0s - loss: 0.7482 - acc: 0
## 55/67 [=======================>......] - ETA: 0s - loss: 0.7035 - acc: 0
## 67/67 [==============================] - 0s 4ms/step - loss: 0.6851 - ac
## Epoch 3/75
```

RNN for time-series data

```
y_test = arframe[~istrain].log_volume

kpred = model.predict(xrnn[~istrain,:,:])
```

```
##
##  1/56 [..............................] - ETA: 8s
## 36/56 [==================>...........] - ETA: 0s
## 56/56 [==============================] - 0s 1ms/step
```

```
np.mean((kpred[:,0]-arframe[~istrain]["log_volume"])**2)
```

```
## 0.6311663306595902
```

```
np.var(y_test)
```

```
## 1.0536865829448807
```

```
1-np.mean((kpred[:,0]-arframe[~istrain]["log_volume"])**2)/np.var(y_test)
```

```
## 0.40099234357185787
```

RNN for time-series data

AR model:

- Layer `Flatten()` simply takes the input sequence and turns it into a long vector of predictors. This results in a linear AR model.
  - To fit a nonlinear AR model, we could add in a hidden layer.

```
from tensorflow.keras.layers import Flatten, Dense
model = Sequential()
model.add(Flatten(input_shape=(5, 3)))
model.add(Dense(units=1))
```

- However, since we already have the matrix of lagged variables from the AR model that we fit earlier, we can actually fit a nonlinear AR model without needing to perform flattening.

```
arframed.iloc[:,1:].columns
```

```
## Index(['L1DJ_return', 'L1log_volume', 'L1log_volatility', 'L2DJ_return',
##        'L2log_volume', 'L2log_volatility', 'L3DJ_return', 'L3log_volume',
##        'L3log_volatility', 'L4DJ_return', 'L4log_volume', 'L4log_volatil',
##        'L5DJ_return', 'L5log_volume', 'L5log_volatility', 'fri', 'mon',
##        'tues', 'wed'],
##       dtype='object')
```

RNN for time-series data

AR model:

- The rest of the steps to fit a nonlinear AR model should by now be familiar.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import RMSprop

x=arframed.iloc[:,1:]

arnnd = Sequential()
arnnd.add(Dense(units=32, activation='relu', input_shape=(x.shape[1],)))
arnnd.add(Dropout(rate=0.5))
arnnd.add(Dense(units=1))

#arnnd.compile(loss='mse', optimizer='rmsprop')
arnnd.compile(loss='mse', optimizer=RMSprop())
```

RNN for time-series data

AR model:

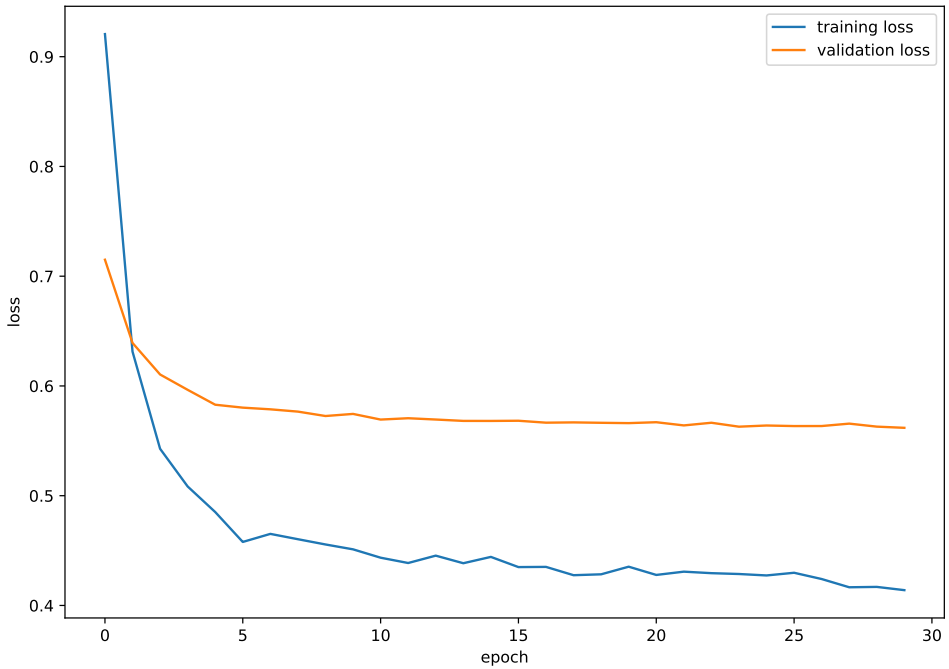
```
import matplotlib.pyplot as plt

history = arnnd.fit(
  x[istrain], arframe[istrain]["log_volume"],
  epochs=30,batch_size=32,
  validation_data=(x[~istrain], arframe[~istrain][ "log_volume"])
  )
```

```
## Epoch 1/30
##
##   1/134 [..............................] - ETA: 34s - loss: 1.2970
##  40/134 [=======>......................] - ETA: 0s - loss: 1.1561
##  82/134 [================>.............] - ETA: 0s - loss: 1.0314
## 123/134 [==========================>...] - ETA: 0s - loss: 0.9470
## 134/134 [==============================] - 1s 3ms/step - loss: 0.9206 -
## Epoch 2/30
##
##   1/134 [..............................] - ETA: 0s - loss: 0.4733
##  26/134 [====>.........................] - ETA: 0s - loss: 0.7572
##  74/134 [==============>...............] - ETA: 0s - loss: 0.6862
## 126/134 [===========================>..] - ETA: 0s - loss: 0.6392
```

RNN for time-series data

AR model:

```
plt.plot(history.history['loss'], label='training loss')
plt.plot(history.history['val_loss'], label='validation loss')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```



RNN for time-series data

```
y_test = arframe[~istrain].log_volume

npred = arnnd.predict(x[~istrain])
```

```
##
##  1/56 [..............................] - ETA: 2s
## 56/56 [==============================] - 0s 873us/step
```

```
np.mean((npred[:,0]-arframe[~istrain]["log_volume"])**2)
```

```
## 0.5618450184540662
```

```
1-np.mean((npred[:,0]-arframe[~istrain]["log_volume"])**2)/np.var(y_test)
```

```
## 0.4667816525823061
```

Summary

- CNNs have had enormous successes in image classification and modeling, and are starting to be used in medical diagnosis. Examples include digital mammography, ophthalmology, MRI scans, and digital X-rays.
- RNNs have had big wins in speech modeling, language translation, and forecasting.

Should we always use deep learning models?

- Often the big successes occur when the signal to noise ratio is high - e.g. image recognition and language translation. Datasets are large, and overfitting is not a big problem.
- For noisier data, simpler models can often work better.
  - On the `NYSE` data, the AR(5) model is much simpler than a RNN, and performed as well.
  - On the `IMDB` review data, the linear model fit did as well as the neural network, and better than the RNN.
- We endorse the `Occam's razor` principal (https://en.wikipedia.org/wiki/Occam%27s_razor) - we prefer simpler models if they work as well. More interpretable!

License