NISS Data Science Essentials for Business Tutorial: *Deep Learning*

Ming Li

Amazon & U. of Washington

Disclaimer: the opinions expressed in this short course is presenter's own and do not represent the view of presenter's employers.

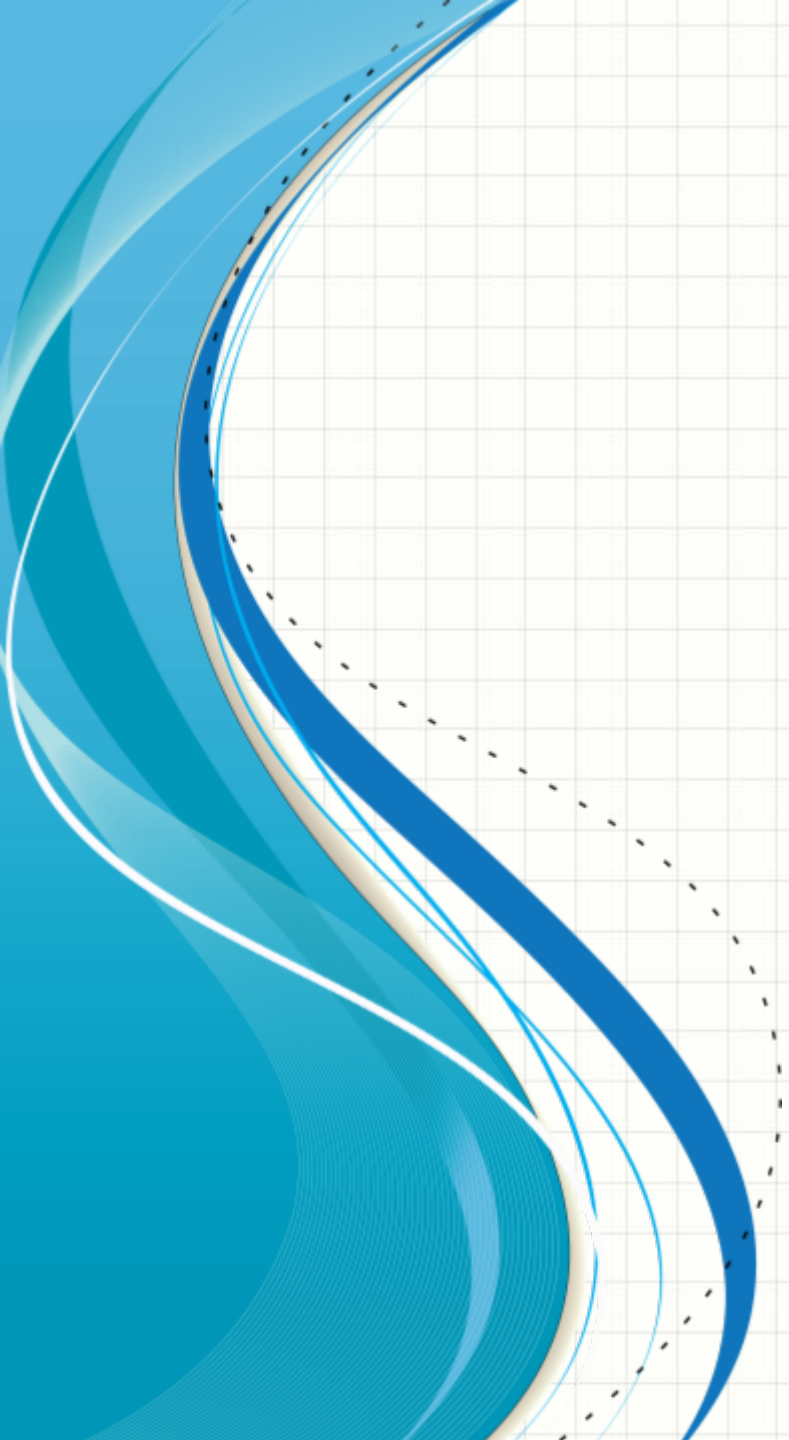
Intended Audience

- Statisticians or practitioners who are exploring data science applications
- Graduate students who are looking for data science opportunities
- University professors who are expanding their data science course scopes
- Scientist or analysts who are trying to deal with un-structured data such as text and image in their day to day work

- *This tutorial is more focusing on introduction of basic concepts and basic applications using R keras. Due to limited time, many of important mathematic details are not covered.*

List of Materials

- Coursera course: Deep Learning Specialization by Andrew Ng et al: a great in-depth introduction of deep learning series
- UFLDL Deep Learning Tutorial: a self-paced tutorial of deep learning basic and applications
- Deep learning with R by François Chollet with J. J. Allaire (a great book of introduction of using keras library in R to train deep learning models, specifically for statisticians)
- Dive into deep learning: interactive deep learning book with code, math and discussion



Data Science Recap

Science Engineering

Physics

Statistics

Electrical Engineering

Statistical Engineering

+

Big Data & Software

=

Data Science

*Statistical engineering

Statis-tician	Data Scientist
Relatively focus on modeling (i.e. science)	Mainly focus on business problem & result (i.e. engineering)
Bring data to model	Bring models to data
Data is relatively small in size and clean in text file formats	Need to work with messy and large amount data in various formats
Usually structured data	Both structured & unstructured data
Usually isolated from production system	Usually embedded in production system

How to Bridge the Gaps?

- **Engineering aspects** of big data, data pipeline and production system:

  R/Python in laptop → Cloud Environment

- **Modeling aspects** to expand toolsets to deal with text and image data more efficiently:

  Traditional machine learning methods (such as regression, random forest, gradient boosting tree, …)

  +

  Deep learning models:

  - ***Standard feedforward neural network for typical data frames***
  - ***Convolutional neural network for image data***
  - ***Word embedding and Recurrent neural network for text data***

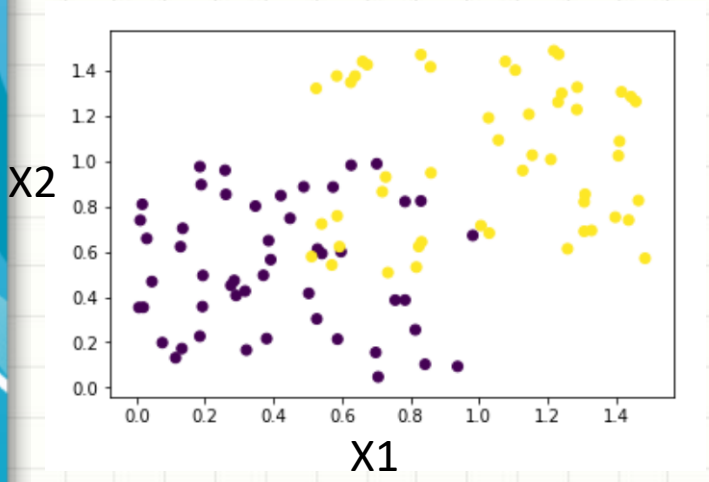
Deep Learning

A Brief Introduction

The New Tool in Data Scientist's Tool Box

A Little Bit of History – Perceptron

- Fun video: https://www.youtube.com/watch?v=cNxadbrN_aI
- Classification of N points into 2 classes: -1 and +1 (i.e. two different colors in the picture below)
- In this example below, only two features to use (X1 and X2)



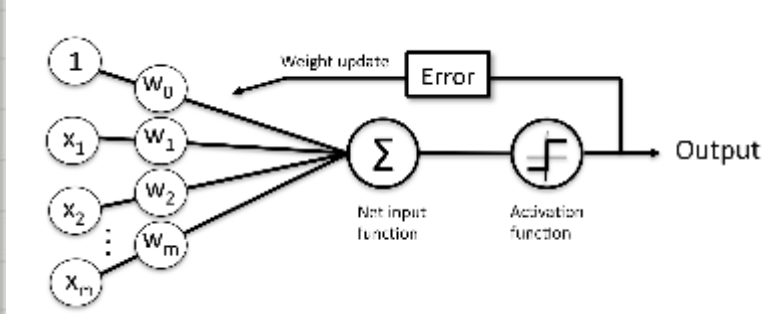
- **Linear functions to separate classes, to find (w_0, w_1, w_2) such that:**

$$\phi_j = w_0 + w_1 x_{1,j} + w_2 x_{2,j}$$

$$Pred_j = \begin{cases} 1, if\ \phi_j > 0 \\ -1, if\ \phi_j \leq 0 \end{cases}$$

- **How to find a good line? Perceptron algorithm:**
  - **Start with random weights**
  - **For each data point**
    1. **Predict class label**
    2. **Update weights when prediction is not correct using a preset learning rate and the value of features of that data point, for example for w_1:**

$$w_1 = w_1 + \eta\left(Actual_j - Pred_j\right)x_{1,j}$$



* From Sebastian *Python Machine Learning*

Key features of perceptron

- It is a linear classification function and the weight is updated after each data points are used to the algorithm (concept similar to stochastic gradient descent).
- The algorithm continues to update when we use the same dataset again and again (i.e. epochs)
- It is not going to converge for none-linearly-spreadable problems.
- We have define training stop criteria such as accuracy of the model or the number of times using the data set to train.

A Little Bit of History – Perceptron

For i in 1:M: **We set a maximum of epochs of M to run.**

For j in 1:N:

$$\phi_j = w_0 + w_1 x_{1,j} + w_2 x_{2,j}$$

$$Pred_j = \begin{cases} 1, if\ \phi_j > 0 \\ -1, if\ \phi_j \leq 0 \end{cases}$$

$$w_0 = w_0 + \eta\left(Actual_j - Pred_j\right)$$
$$w_1 = w_1 + \eta\left(Actual_j - Pred_j\right)x_{1,j}$$
$$w_2 = w_2 + \eta\left(Actual_j - Pred_j\right)x_{2,j}$$

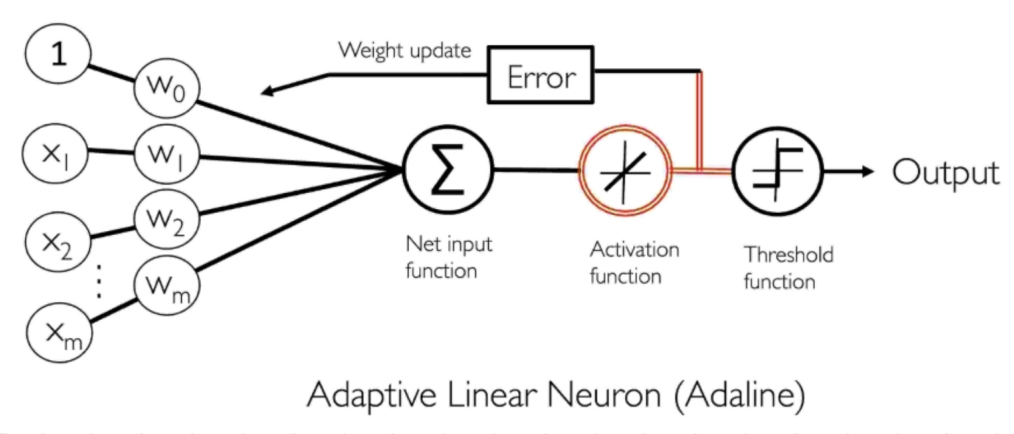
For every data point, we update the weight based on the prediction correctness, learning rate and feature values.

Calculate accuracy for the entire dataset to see whether the criteria has meet.

For not linearly separable dataset, we need to use some accuracy threshold or number of epochs to stop the algorithm.

Perceptron R notebook: link

A Little Bit of History – Adaline



Adaptive Linear Neuron (Adaline)

* From Sebastian *Python Machine Learning*

Adaline R notebook: link

- We can use the error for the entire dataset as the loss function (i.e. SSE): $\sum_{j=1}^{N}\left(Actual_j - \phi_j\right)^2$, and update the weight using gradient descent.
- If we use a logistic function as the activation, then it becomes *logistic regression*.

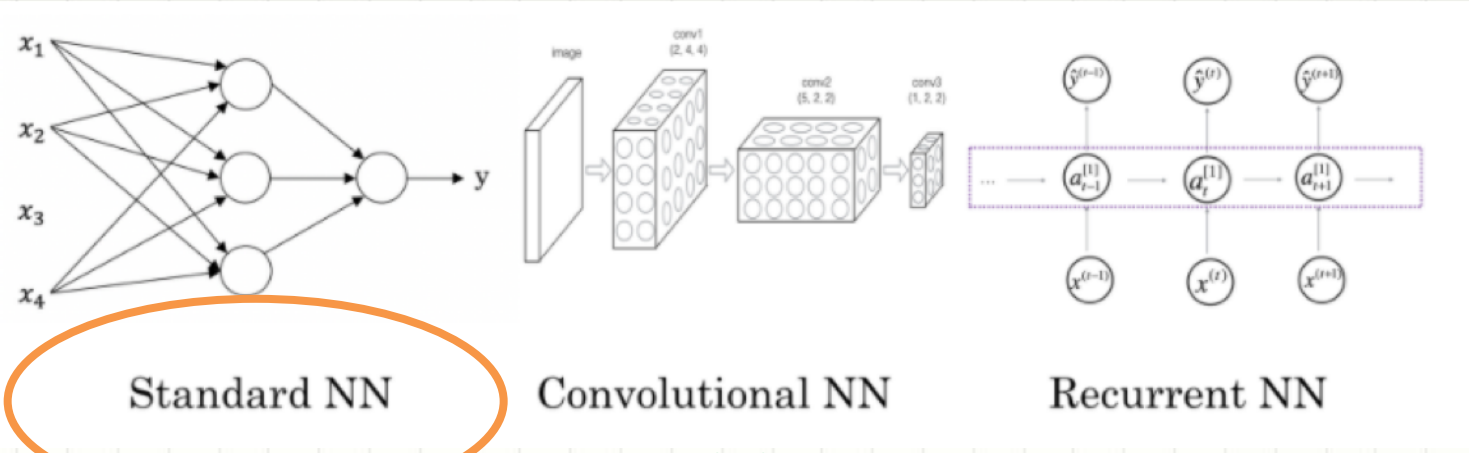
- Very similar to Perceptron and the only difference is that the error is calculated based on $\frac{1}{2}\left(Actual_j - \phi_j\right)^2$, i.e. the square error.
- With the square error, which is differentiable, now we can use stochastic gradient descent method to update the weights:

$$w_0 = w_0 + \eta\left(Actual_j - \phi_j\right)$$
$$w_1 = w_1 + \eta\left(Actual_j - \phi_j\right)x_{1,j}$$
$$w_2 = w_2 + \eta\left(Actual_j - \phi_j\right)x_{2,j}$$

- When calculating prediction accuracy, we still based on whether ϕ_j is larger than zero or not with the final weight learned from the data.

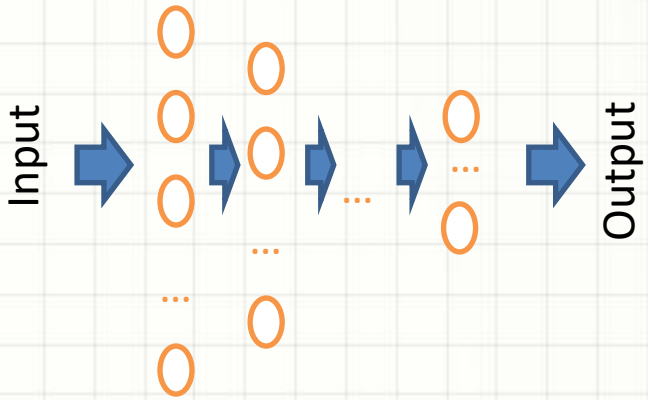
Types of NNs

** Figure adapted from slides by Andrew NG:*
Deep Learning Specialization



2D, 3D or higher dimension data

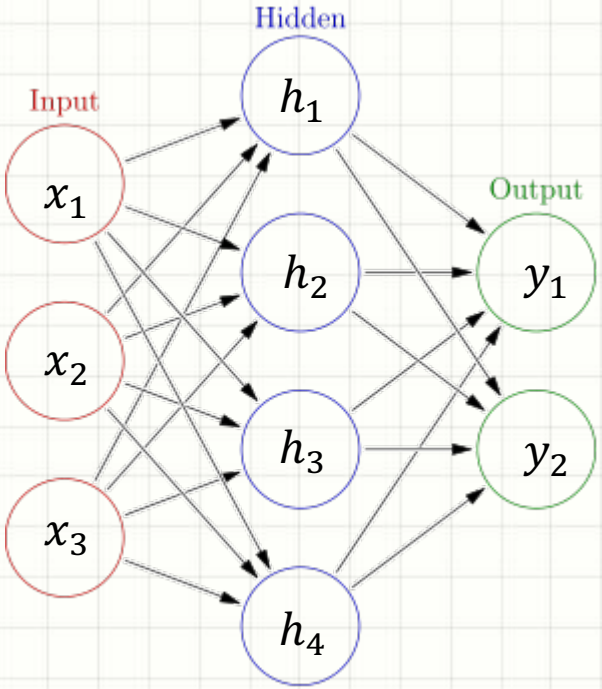
A direct extension of Adaline is to add more neurons in a layer, non-linear activation function, and add more intermedia layers to make it a feed forward neural network.



Each neuron is fully connected to the neurons on the next layer.

Feed Forward Neural Network

Simple Feed Forward Neural Network (FFNN)



Concepts stated as early as 1940s!

$$h_j = f_1\left(b_{j0} + \sum_{i=1}^{3} b_{ji}x_i\right), j = 1,2,3,4$$

$$y_k = f_2\left(b_{k0} + \sum_{j=1}^{4} b_{kj}h_j\right), k = 1,2$$

$$\boldsymbol{y} = f_2(f_1(\boldsymbol{x}, \boldsymbol{b}_1), \boldsymbol{b}_2)$$ ***Vector representation***

$[x_1, x_2, x_3]$: Input features vector for one observation

$[y, y_2\ ]$: Actual output outcome for one observation

$f_1(\cdot), f_2(\cdot)$: Activation functions, usually non-linear

$L(y, \hat{y}(\boldsymbol{b}, \boldsymbol{x})\ )$: Loss function where $\hat{y}$ is the model forecast responses and y is actual observed responses

Total number of parameters (i.e. size of a NN): (3+1)*4 + (4+1)*2 = 26

Typical Loss Functions

- Two-class binary responses
  - ✓ Binary cross-entropy:

$$\sum_{i=1}^{N}\left(-y_i log(p_i) - (1-y_i) log(1-p_i)\right)$$

where y_i is actual value of 1 or 0, p_i is the predicted probability of being 1, and N is the total number of observations in the training data.

- Multiple-class responses
  - ✓ Categorical cross-entropy for M classes:

$$\sum_{i=1}^{N}\left(-\sum_{j=1}^{M} y_{i,j} log(p_{i,j})\right)$$

where $y_{i,j}$ is actual value of 1 or 0 for a class of j, $p_{i,j}$ is the predicted probability of being at class j and N is the total number of observations in the training data.

- Continuous responses
  - ✓ Mean square error
  - ✓ Mean absolute error
  - ✓ Mean absolute percentage error

From Slow Progress to Wide Adoption

1940s – 1980s, very slow progress due to:

- ❑ **Computation hardware capacity limitation**
- ❑ **Number of observations with labeled results in the dataset**
- ❑ **Lack efficient algorithm to estimate the parameters in the model**

1980s – 2010, a few applications in real word due to:

- ❑ **Moore's Law + GPU**
- ❑ **Internet generated large labeled dataset**
- ❑ **Efficient algorithm for optimization (i.e. SGD + Backpropagation)**
- ❑ **Better activation functions (i.e. Relu)**

2010-Now, very broad application in various areas:

- ❑ **Near-human-level image classification**
- ❑ **Near-human-level handwriting transcription**
- ❑ **Much improved speech recognition**
- ❑ **Much improved machine translation**

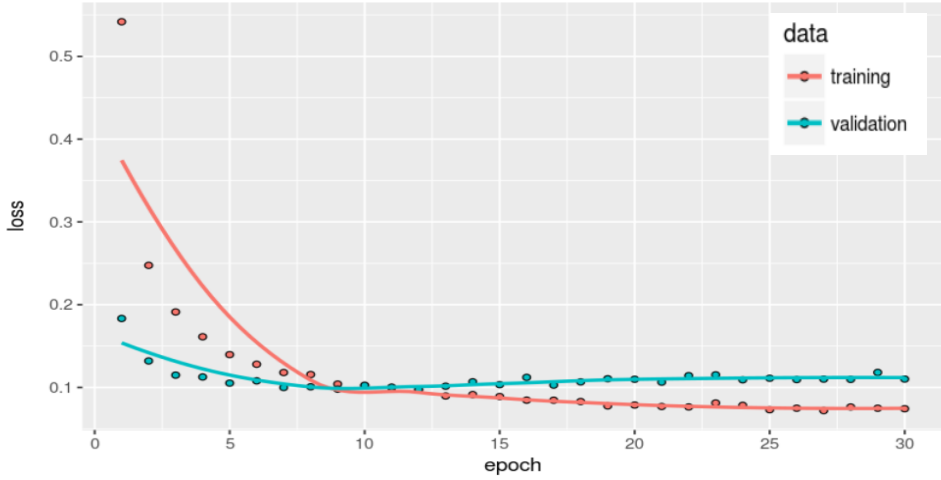
Now we know working neural network models usually contains many layers (i.e. the depth of the network is deep), and to achieve near-human-level accuracy the deep neural network need huge training dataset (for example millions of labeled pictures for image classification problem).

Optimization Methods

- Mini-batch Stochastic Gradient Descent (SGD)
  - Use **a small segment of data** (i.e. 128 or 256) to update the model parameters: $b = b - \alpha \nabla_b L(b, x^m, y^m)$ where α is the **learning rate** which is a hyper parameter that can be changed, and m is the mini-batch.
  - Gradients are efficiently calculated using **backpropagation** method
  - When the entire dataset are used to updated the SGD, it is called one **epoch** and multiple epochs are needed to run to reach convergence
  - An updated version with 'momentum' for quick convergence:

$$v = \gamma v_{last} + \alpha \nabla_b L(b, x^m, y^m)$$
$$b = b - v$$

  - The optimal number for epoch is determined by when the model is not overfitted (i.e. validation accuracy reaches the best performance).



More Optimization Methods

- With Adaptive Learning Rates
  - **Adagrad**: Scale learning rate inversely proportional to the square root of the sum of all historical squared values of the gradient (stored for every weight)
  - **RMSProp**: Exponentially weighted moving average to accumulate gradients
  - **AdaDelta**: Uses sliding window to accumulate gradients
  - **Adam** (adaptive moments): momentum integrated, bias correction in decay
- A good summary: http://ruder.io/optimizing-gradient-descent/

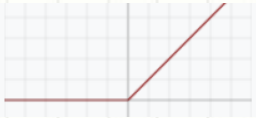
Activation Functions

$$\boldsymbol{y} = f_n(f_{n-1}(f_{n-2}(\dots f_1(\boldsymbol{x}, \boldsymbol{b}_1) \dots, \boldsymbol{b}_{n-2}), \boldsymbol{b}_{n-1}), \boldsymbol{b}_n)$$

Gradient: $$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{b_1}} = \frac{\partial \boldsymbol{y}}{\partial f_n} \frac{\partial f_n}{\partial f_{n-1}} \frac{\partial f_{n-2}}{\partial f_{n-1}} \dots \frac{\partial f_1}{\partial \boldsymbol{b_1}}$$

- Intermediate layers
  - Relu (i.e. rectified linear unit) is usually a good choice which has the following good properties: (1) fast computation; (2) non-linear; (3) reduced likelihood of the gradient to vanish; (4) Unconstrained response
  - Sigmoid, studied in the past, not as good as Relu in deep learning, due to the gradient vanishing problem when there are many layers
- Last layer which connects to the output
  - Binary classification: sigmoid with binary cross entropy as loss function
  - Multiple class, single-label classification: softmax with categorical cross entropy for loss function
  - Continuous responses: identity function (i.e. y = x)

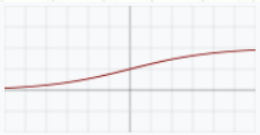
Rectified linear unit (ReLU)[10]



$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

Softmax $$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \quad \text{for } i = 1, \dots, J$$

Logistic (a.k.a. Sigmoid or Soft step)



$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

* Graphs from wiki: link

Deal With Overfitting

- Huge number of parameters, even with large amount of training data, there is a potential of overfitting
  - Overfitting due to size of the NN (i.e. total number of parameters)
  - Overfitting due to using the training data for too many epochs
- Solution for overfitting due to NN size
  - Dropout: randomly dropout some proportion (such as 0.3 or 0.4) of nodes at each layer, which is similar to random forest concept
  - Using L1 or L2 regularization in the activation function at each layer
- Solution for overfitting due to using too many epochs
  - Run NN with large number of epochs to reach overfitting region through cross validation from training/validation vs. epoch curve
  - Choose the model with number of epochs that have the minimum validation accuracy as the final NN model (i.e. early stop)

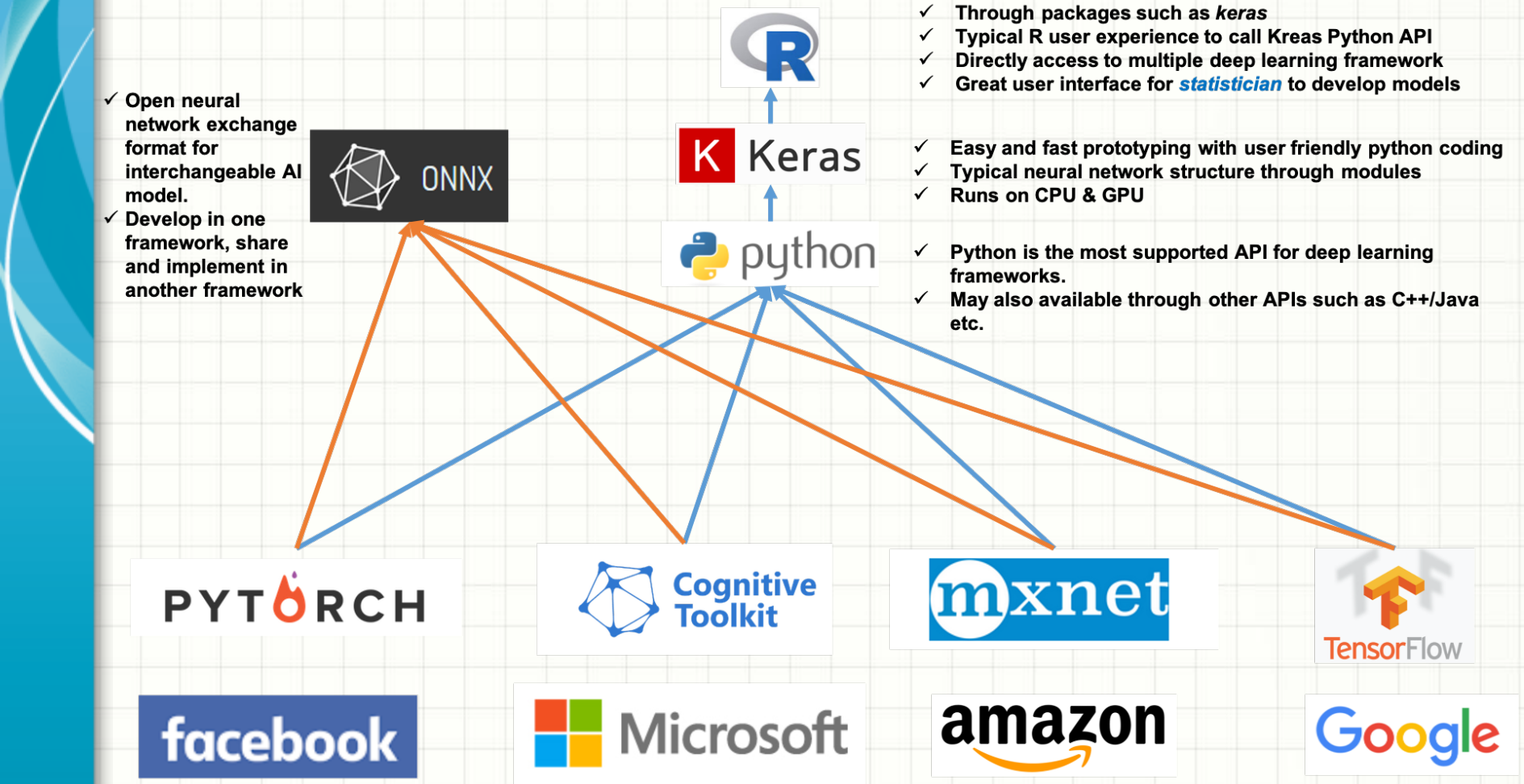
Recap of A Few Key Concepts

- **Data**: Require large well-labeled dataset
- **Computation**: intensive matrix-matrix operation
- **Structure** of fully connected feedforward NN
  - Size of the NN: total number of parameters
  - Depth: total number of layers (this is where deep learning comes from)
  - Width of a particular layer: number of nodes (i.e. neurons) in that layer
- **Activation function**
  - Intermediate layers
  - Last layer connecting to outputs
- **Loss function**
  - Classification (i.e. categorical response)
  - Regression (i.e. continuous response)
- **Optimization methods (SGD)**
  - Batch size
  - Learning rate
  - Epoch
- **Deal with overfitting**
  - Dropout
  - Regularization (L1 or L2)

Deep Learning Across Platforms

Luckily, there is no needs to write our own functions from scratch! There are thousands of dedicated developers, engineers and scientists are working on open source deep learning frameworks.

Open NN Exchange Format (ONNX)

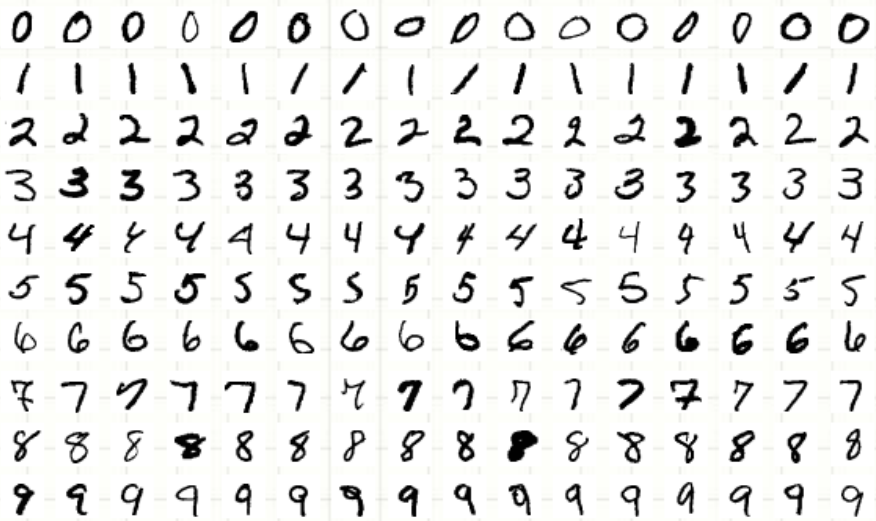


https://onnx.ai/index.html

The MNIST Dataset

MNIST Dataset

- Originally created by NIST, then modified for machine leaning training purpose
- Contains 70000 handwritten digit images and the label of the digit in the image where 60000 images are the training dataset and the rest 10000 images are the testing dataset.
- Census Bureau employees and American high school students wrote these digits
- Each image is 28x28 pixel in greyscale
- Yann LeCun used convolutional network LeNet to achieve < 1% error rate at 1990s
- More details: https://en.wikipedia.org/wiki/MNIST_database



Using Keras R Package To Build Feed Forward Neural Network Model

Procedures

- Data preprocessing (from image to list of input features)
  - One image of 28x28 grey scale value matrix → 784 column of features
  - Scale the value to between 0 and 1, by divide each value by 255
  - Make response categorical (i.e. 10 columns with the corresponding digit column 1 and rest columns zero.
- Load keras package and build a neural network with a few layers
  - Define a placeholder object for the NN structure
  - 1st layer using 256 nodes, fully connected, using 'relu' activation function and connect from the input 784 features
  - 2nd layer using 128 nodes, fully connected, using 'relu' activation function
  - 3rd layer using 64 nodes, fully connected, using 'relu' activation function
  - 4th layer using 10 nodes, fully connected, using 'softmax' activation function and connect to the output 10 columns
  - Add drop out to the first three layers to prevent overfitting
- Compile the NN model, define loss function, optimizer, and metrics to follow
- Fit the NN model using the training dataset, define epoch, mini batch size, and validation size used in the training where the metrics will be checked
- Predict using the fitted NN model using the testing dataset

R Scripts

```r
dnn_model <- keras_model_sequential()
dnn_model %>%
  layer_dense(units = 256, activation = 'relu', input_shape = c(784)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 64, activation = 'relu') %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = 'softmax')
```

Define NN Structure

```r
dnn_model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)
```

Compile and define loss function, optimizer and metrics to monitor during the training

```r
dnn_history <- dnn_model %>% fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2
)
```

Fit the model using training dataset and define epochs, batch size and validation data

```r
dnn_pred <- dnn_model %>%
              predict_classes(x_test)
```

Predict new outcomes using the trained model

Size of the Model

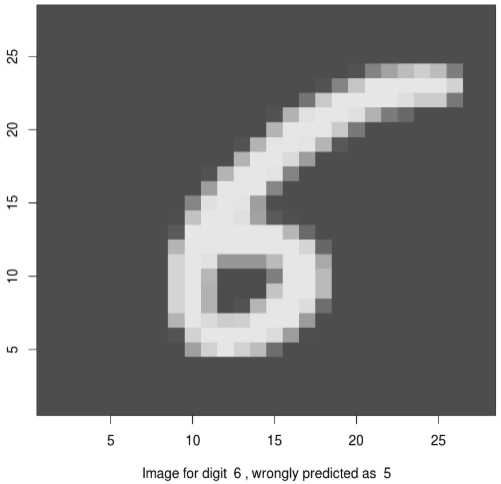
```
1 summary(dnn_model)
```

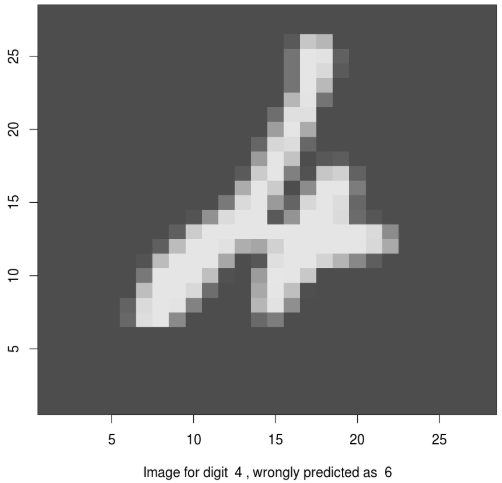
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 256)               200960
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               32896
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                8256
_________________________________________________________________
dropout_3 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                650
=================================================================
Total params: 242,762
Trainable params: 242,762
Non-trainable params: 0
_________________________________________________________________
```

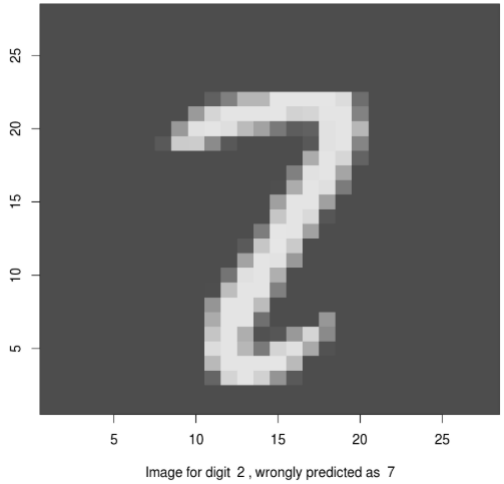
Performance

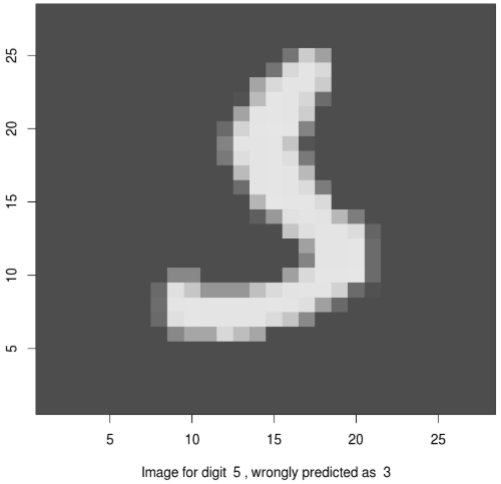
Accuracy: 98% without much fine tuning

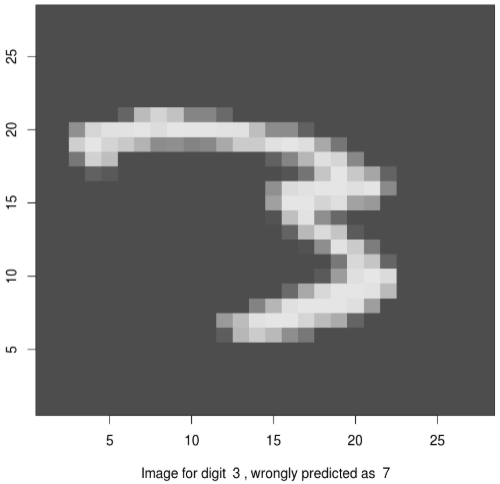
A few misclassified images:

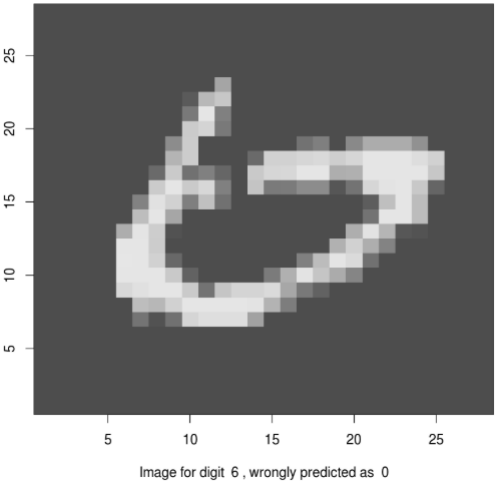




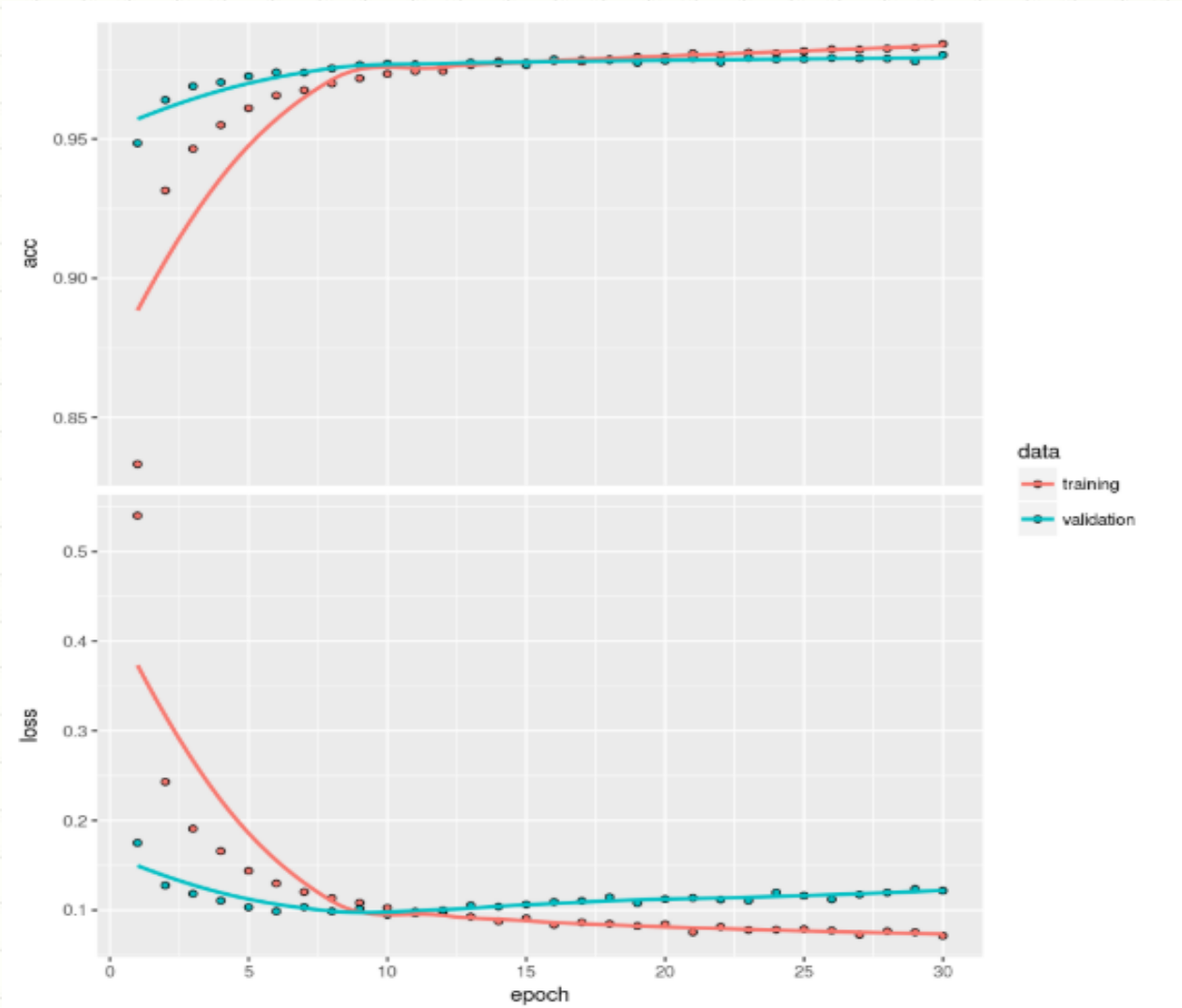








Cross-Validation Curve



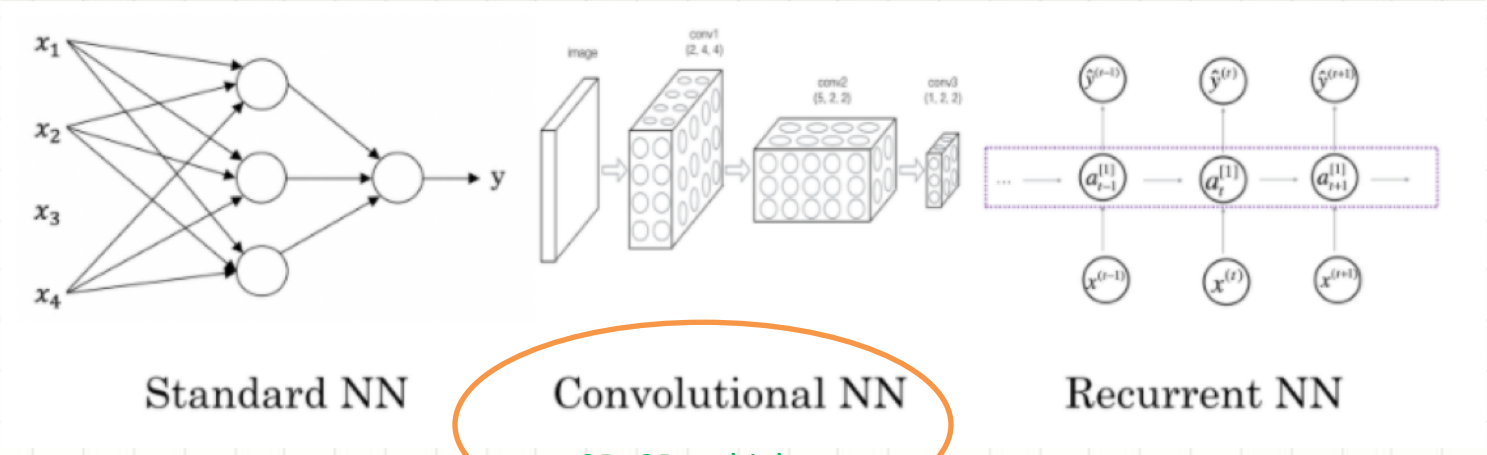
FFNN Hands-on Session

Databrick Community Edition: link
FFNN Notebook: link

Convolutional Neural Network

Types of NNs

** Figure adapted from slides by Andrew NG:*
Deep Learning Specialization



2D, 3D or higher dimension data

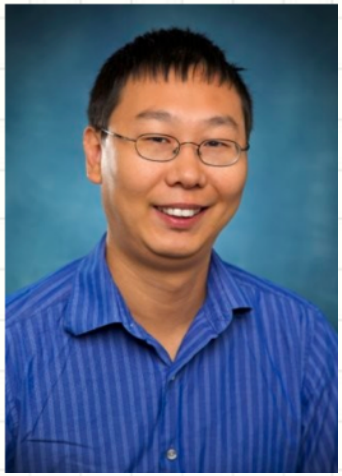
There are **many different** CNN structures and applications. In this course, we focus on the application of *image classification* of the handwritten digits same as FFNN part.

Image as Numbers

Grey Scale
1 channel

Each pixel is an integer between 0 and 255

23	45	1	3
67	87	102	103
23	56	234	89
4	108	76	44



Color Picture
3 channel: R G B

Each pixel is three integers between 0 and 255

		66	104	8	10
	156	78	23	4	78
201	223	109	78	178	66
23	45	221	178	56	23
99	83	76	105	77	
132	15	6	98		

There are more ways to describe pictures.

Concept of Convolution

Input (5x5) Kernel/Filter (3x3) Output (3x3)

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

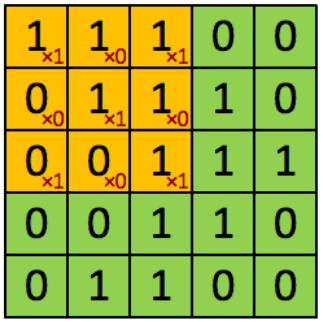
*

1	0	1
0	1	0
1	0	1

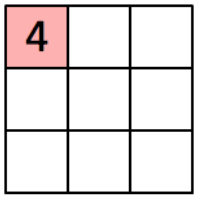
=

4	3	4
2	4	3
2	3	4

With stride 1 (move 1 pixel at a time)



Image



Convolved Feature

For $n \times n$ input picture, $f \times f$ kernal, and stride s, the output after convolution will has size:

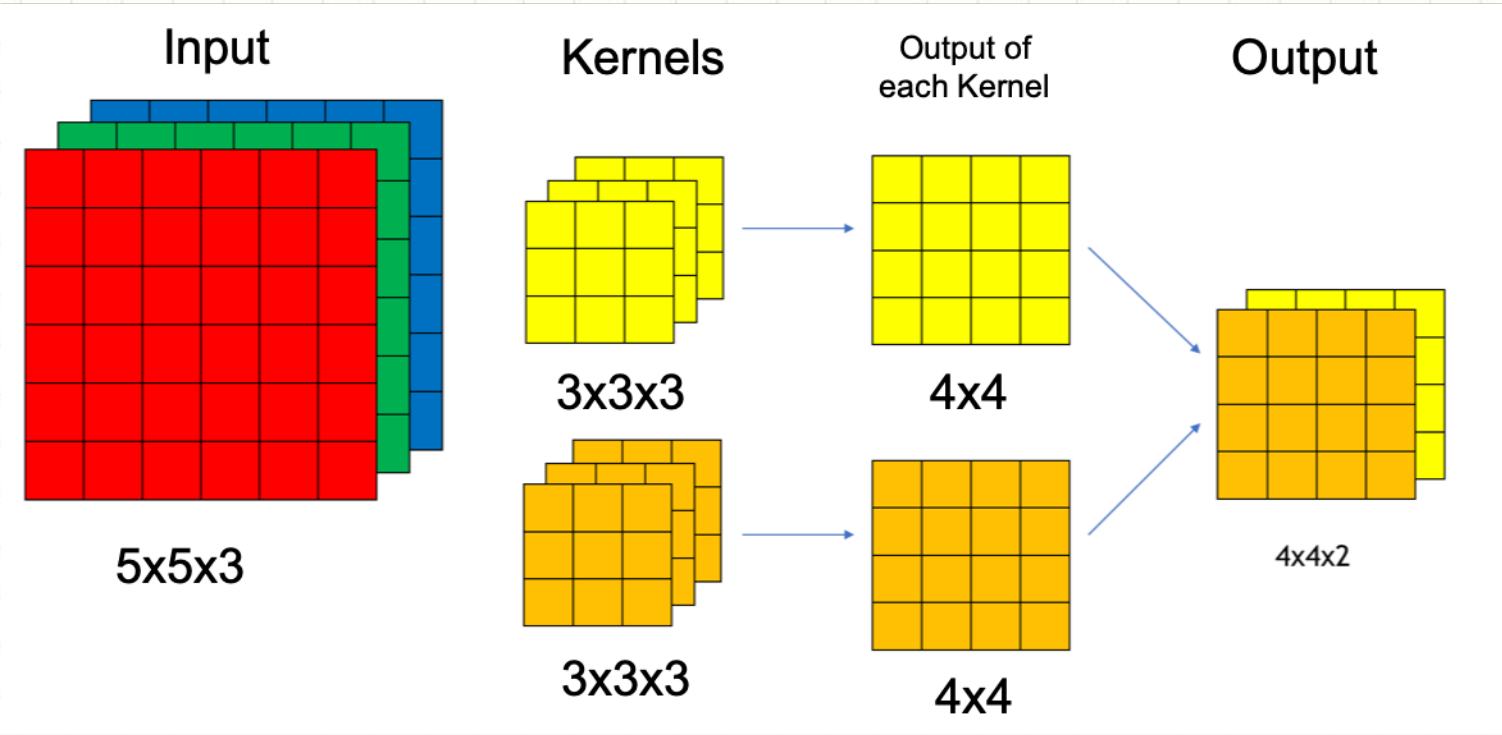
$$\left(\frac{n-f}{s}+1\right) \times \left(\frac{n-f}{s}+1\right)$$

*** Animation graph from UFLDL Tutorial: link**

Extend to Multiple Channels and Filters

For $5\times5\times3$ input RGB picture, we will need $3\times3\times3$ as one kernel/filter. We apply the same procedure for each 5×5 input and 3×3 filter, and get 3 intermedia matrixes of 4×4. The final output after convolution for one set of kernel will has size: 4×4 by applying element-wise average.

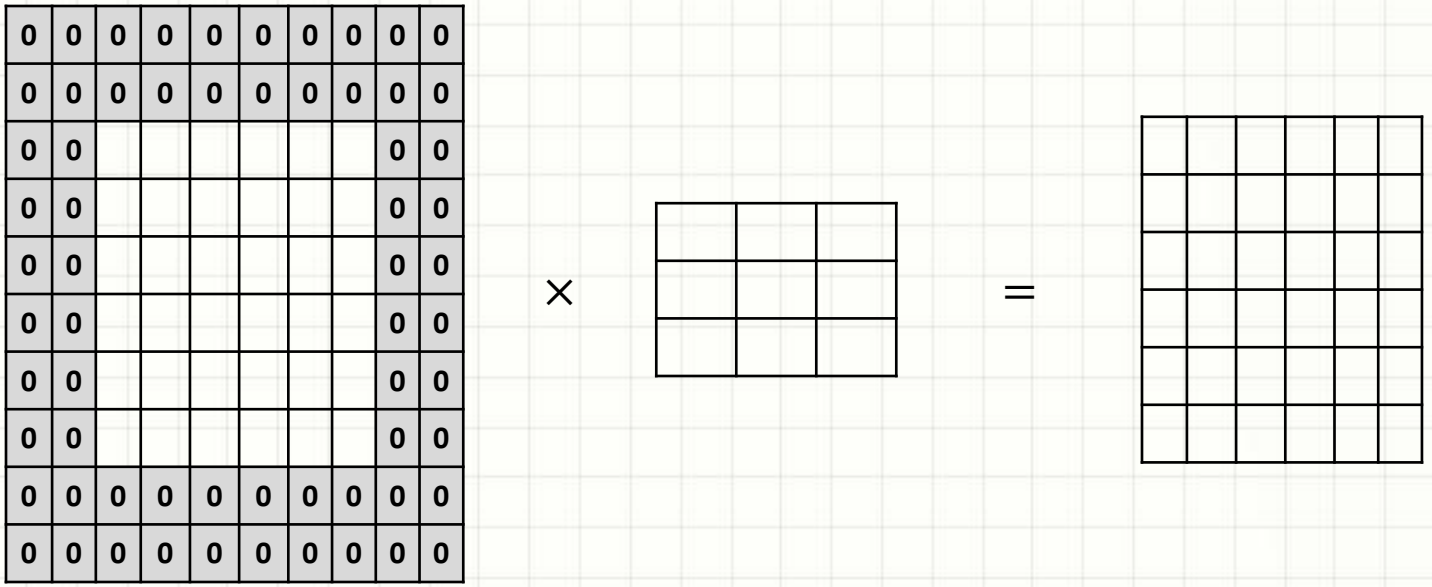
We can apply multiple channels to the same input, in the graph below, there are two set of $3\times3\times3$ kernels. The final output will be two 4×4 matrixes (i.e. a tensor).



** Figure adapted from slides by Andrew NG: Deep Learning Specialization*

Padding

Convolution will reduce the size of the matrix after each convolution layer. Sometime, we want to keep the output dimension to be the same as input dimension. It can be done by padding (i.e. put zeroes at the edges of the input matrix).



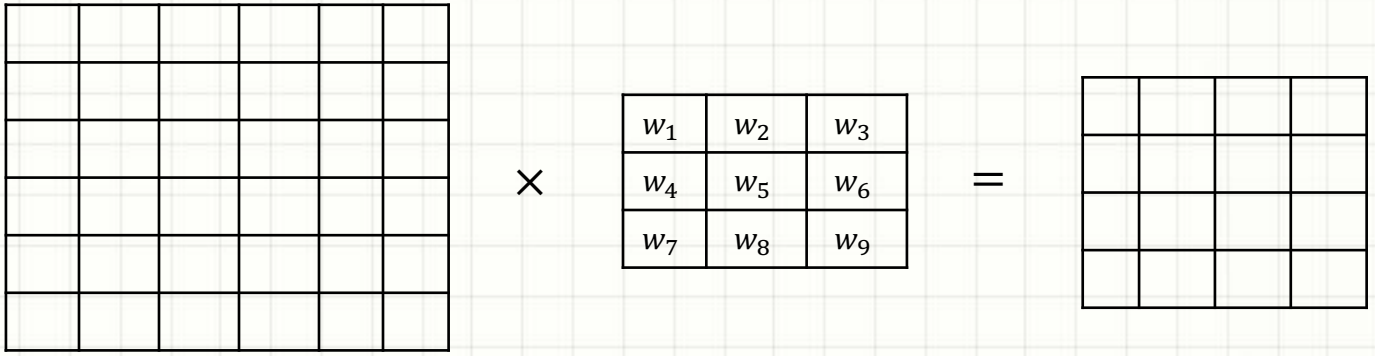
For $n \times n$ input picture, $f \times f$ kernal, stride s, and padding p, the output after convolution will has size:

$$\left(\frac{n+2p-f}{s}+1\right) \times \left(\frac{n+2p-f}{s}+1\right)$$

Estimation Process

Similar to FFNN, we can define number of convolution layers and size & number of kernels to use at each layer to construct a CNN. The CNN estimation process is to get a set of values for all the kernels to minimize the loss function using *mini-batch gradient descent*. For MNIST dataset, the loss function is the same as what we defined in FFNN example. Due to the time limit, we will not go into detail of the estimation process through back-propagation. More details can be found: link

In the picture below, it shows one 3x3 kernel with unknow parameters $w_1, \dots, w_9$:



6x6 × 3x3 = 4x4

If layer l is a convolution layer:

- $f^{[l]}$ = filter size
- $p^{[l]}$ = padding
- $s^{[l]}$ = stride
- $n_C^{[l]}$ = number of filters

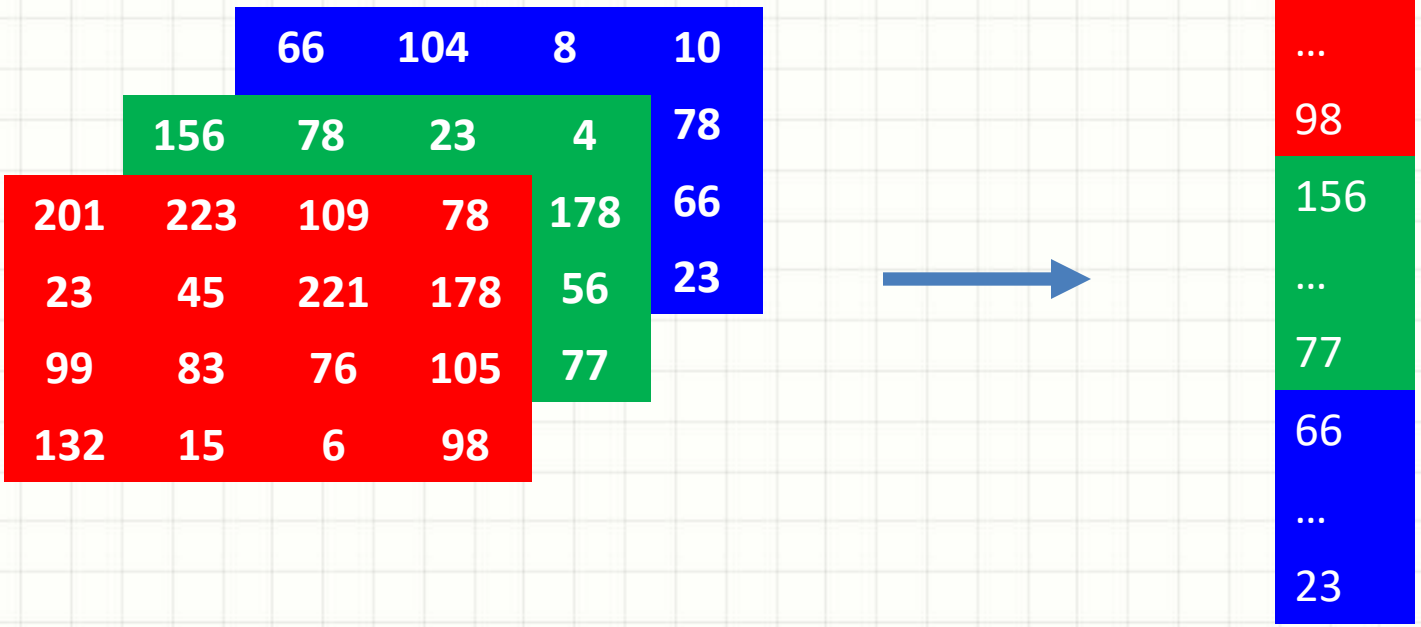
- Input: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$
- Output: $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$
- $n_H^{[l]} = \frac{n_W^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1$ (similar for W)

- Each filter: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]}$
- Activations: $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$
- Weights: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]} \times n_C^{[l]}$
- bias: $b_C^{[l]}$

Flatten Layer

If we only have convolution layers, the output will be a tensor. How can we connect a tensor to the output of 10 classes in the MNIST example? We need to *flatten the tensor into a vector*. The graph below shows a flatten operation to convert a 4x4x3 tensor into a vector of 48 elements. For flatten layer, there is *no parameters to be estimated*, just change the shape.

Once it is flattened, we can add a few fully connected layers before finally connecting to the output layer of 10 classes for the NIST example.



Pooling Layer

One way to reduce the dimension of matrixes. A $f \times s$ rectangular area is replaced by one single number by taking the *maximum* or *average* of these $f \times s$ values.

34	**3**	**21**	**7**
55	**12**	**78**	**124**
54	**7**	**76**	**66**
21	**15**	**23**	**210**

Input
4x4

$f = 2$
$s = 2$

Max Pooling

55	**124**
54	**210**

Output
2x2

Average Pooling

26	**57.5**
24.25	**93.75**

Output
2x2

CNN Example

With convolution layers, pooling layers, flatten layer, and fully connect layers, we can create functional CNN structure to connect inputs and output for image related applications.

LeNet-5 Architecture

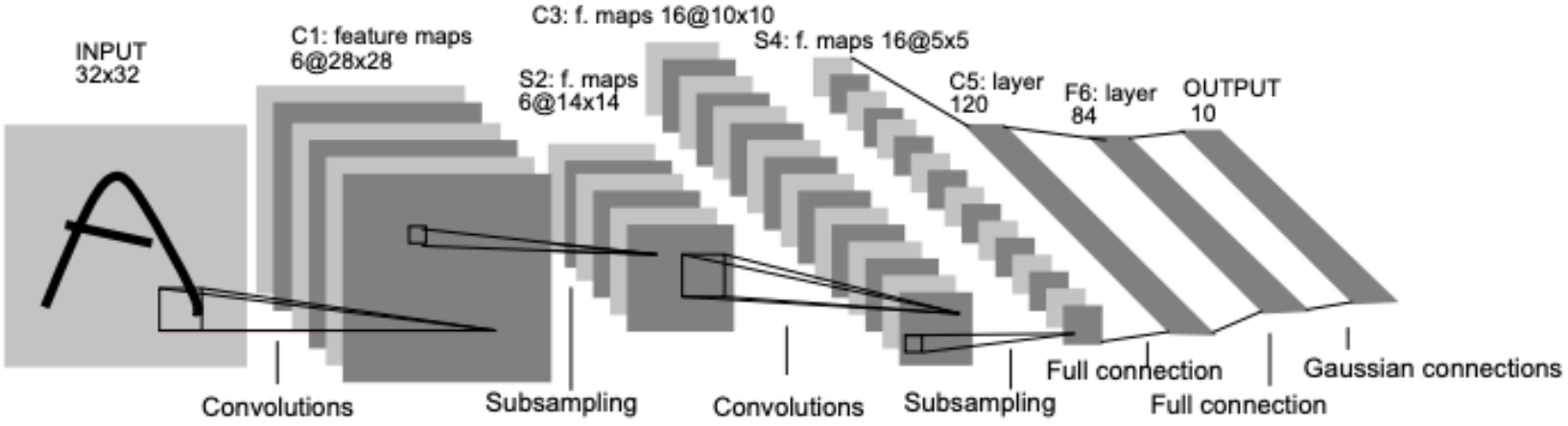


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

LeCun et al., 1998. Gradient-based learning applied to document recognition. Link, Link

More modern CNN structures can be found: link

Create and Train CNN Model in Keras

```
# Define model structure
cnn_model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3),
  activation = "relu", input_shape = input_shape) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_dropout(rate = 0.25) %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = num_classes, activation = "softmax")
```

```
# Compile model
cnn_model %>% compile(
  loss = loss_categorical_crossentropy,
  optimizer = optimizer_adadelta(),
  metrics = c('accuracy')
)
```

```
# Train model
cnn_history <- cnn_model %>%
  fit(
    x_train, y_train,
    batch_size = batch_size,
    epochs = epochs,
    validation_split = 0.2
  )
```

```
> summary(cnn_model)
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
conv2d_2 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 64)	0
dropout_1 (Dropout)	(None, 12, 12, 64)	0
flatten_1 (Flatten)	(None, 9216)	0
dense_1 (Dense)	(None, 128)	1179776
dropout_2 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 10)	1290

Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0

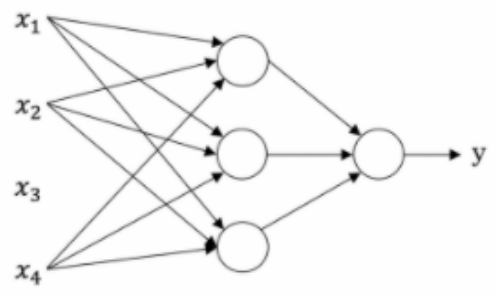
CNN Hands-on Session

CNN Notebook: link

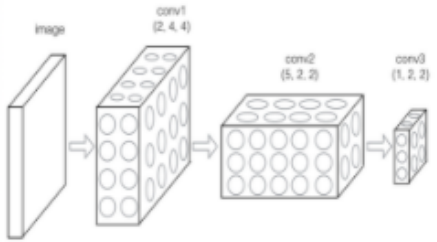
Recurrent Neural Network

Types of NNs

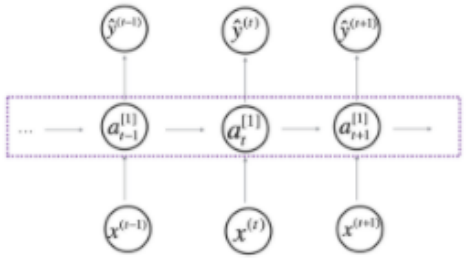
** Figures adapted from slides by Andrew NG:*
Deep Learning Specialization



Standard NN

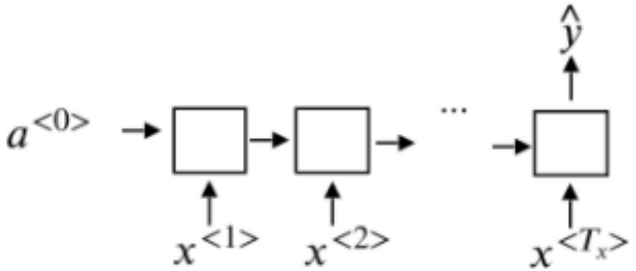


Convolutional NN



Recurrent NN

Sequence is key!



Many to one

There are **many different** RNN structures and applications. In this course, we focus on one type of application called **many to one** problem (i.e. the input is a sequence of words and the output is just one variable).

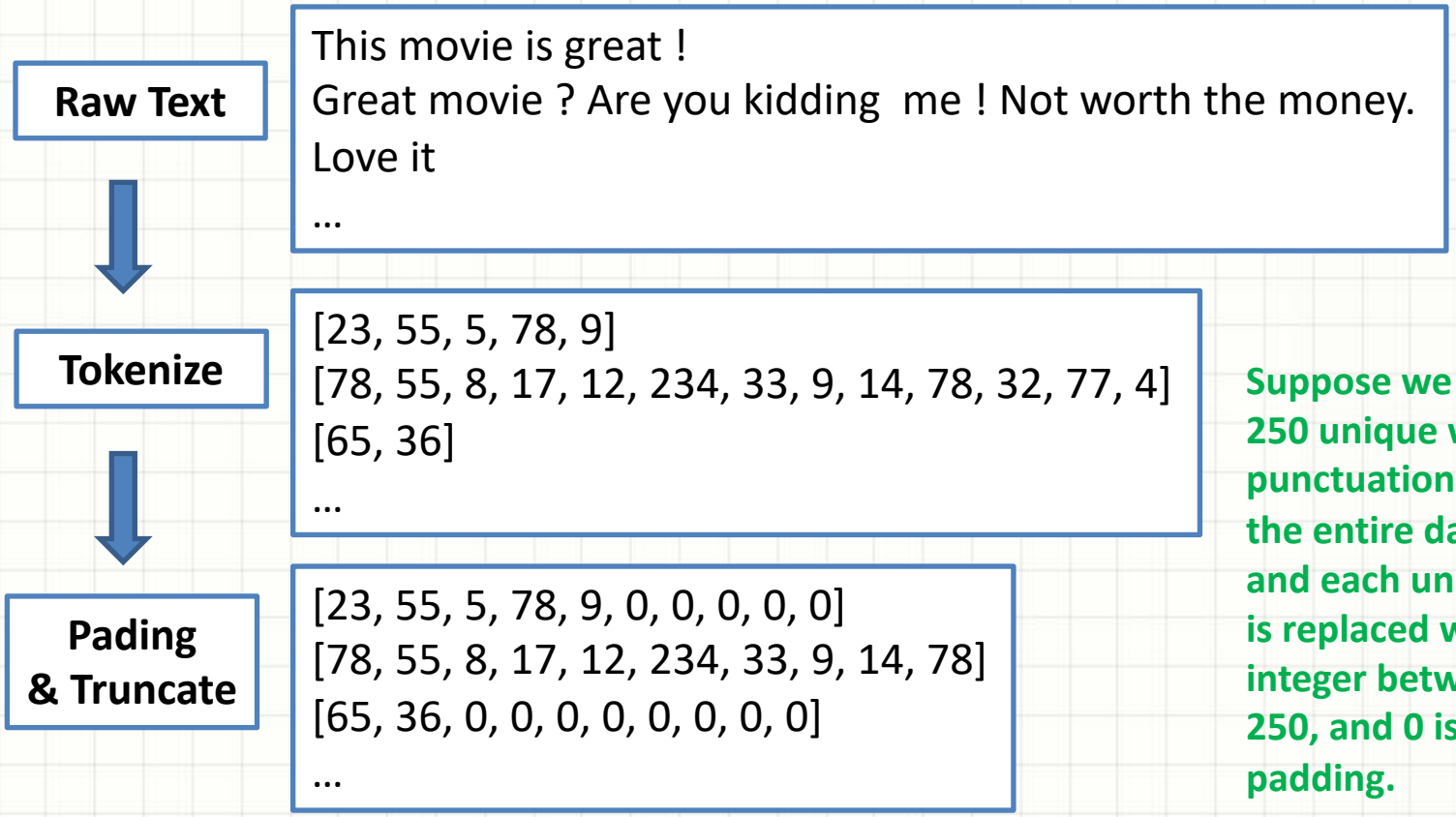
Problem Description & IMDB dataset

- **Raw data: 50,000 movie review text (X) and it's corresponding sentiment of positive (1, 50%) or negative (0, 50%) (Y).**
- **Included in keras package, can be easily loaded and processed.**
- **Task: train a model such that we can use review text as input to predict whether its sentiment is positive or negative (i.e. binary classification problem).**
- **It is one example of many-to-one RNN application.**
- **The raw data can be found here. Sample data in its text format:**

Input X (Review)	Output Y (Sentiment)
Very smart, sometimes shocking, I just love it. It shoved one more side of David's brilliant talent. He impressed me greatly! David is the best. The movie captivates your attention for every second.	Positive (i.e. 1)
I and a friend rented this movie. We both found the movie soundtrack and production techniques to be lagging. The movie's plot appeared to drag on throughout with little surprise in the ending. We both agreed that the movie could have been compressed into roughly an hour giving it more suspense and moving plot.	Negative (i.e. 0)

Analyzing Text - Tokenization

Algorithm cannot deal with raw text and we have to convert text into numbers for machine learning methods.



Suppose we only have 250 unique words and punctuation marks in the entire dataset, and each unique word is replaced with an integer between 1 to 250, and 0 is used for padding.

Now we have a typical data frame, each row is an observation, and each column is a feature. Here we have 10 columns by designing after the padding and truncating stage. We have converted raw text into categorical integers.

Analyzing Text – Encoding/Embedding

Categorical integers can not be used directly to algorithm as there is no mathematical relationship among these categories. We have to use either Encoding or Embedding.

One Hot Encoding

One_Column

23

250 columns!

One_Column_dummy

0, 0, 0, …, 1, 0, 0, …, 0, 0

Original data frame

```
[23, 55, 5, 78, 9, 0, 0, 0, 0, 0]
[78, 55, 8, 17, 12, 234, 33, 9, 14, 78]
[65, 36, 0, 0, 0, 0, 0, 0, 0, 0]
…
```

10 columns

OHE data frame

```
[0, 0, 0, …, 1, 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]
…
```

250x10 columns!

The OHE will explode the dimension if we have 10,000 unique words in vocabulary and a few hundreds words in the each review in the training dataset!

It is binary, sparse, without consider the relationship among words. OHE is generally not used in text related models.

Dense Embedding

One_Column

23

Configurable (here use 4)

Embedding_Column

0.3, 0.9, 0.1, 0.2

Original data frame

```
[23, 55, 5, 78, 9, 0, 0, 0, 0, 0]
[78, 55, 8, 17, 12, 234, 33, 9, 14, 78]
[65, 36, 0, 0, 0, 0, 0, 0, 0, 0]
…
```

10 columns

OHE data frame

```
[0.3, 0.9, 0.1, 0.2, …, 0.7, 0.8]
[0.2, 0.7, 0.3, 0.7, …, 0.4, 0.3]
[0.5, 0.8, 0.4, 0.6, …, 0.5, 0.9]
…
```

10x4 columns!

Dense embedding utilizes the inherit relationship of words and dramatically reduce the embedded dimension. The dimension is a vector space and can be configured such as 4 for this example. Word2vec has 300 for the vector.

It is low dimension compared with OHE, real number, meaningful, and can be learned on specific data or use pre-trained embeddings.

Analyzing Text – Pre-Trained Embeddings

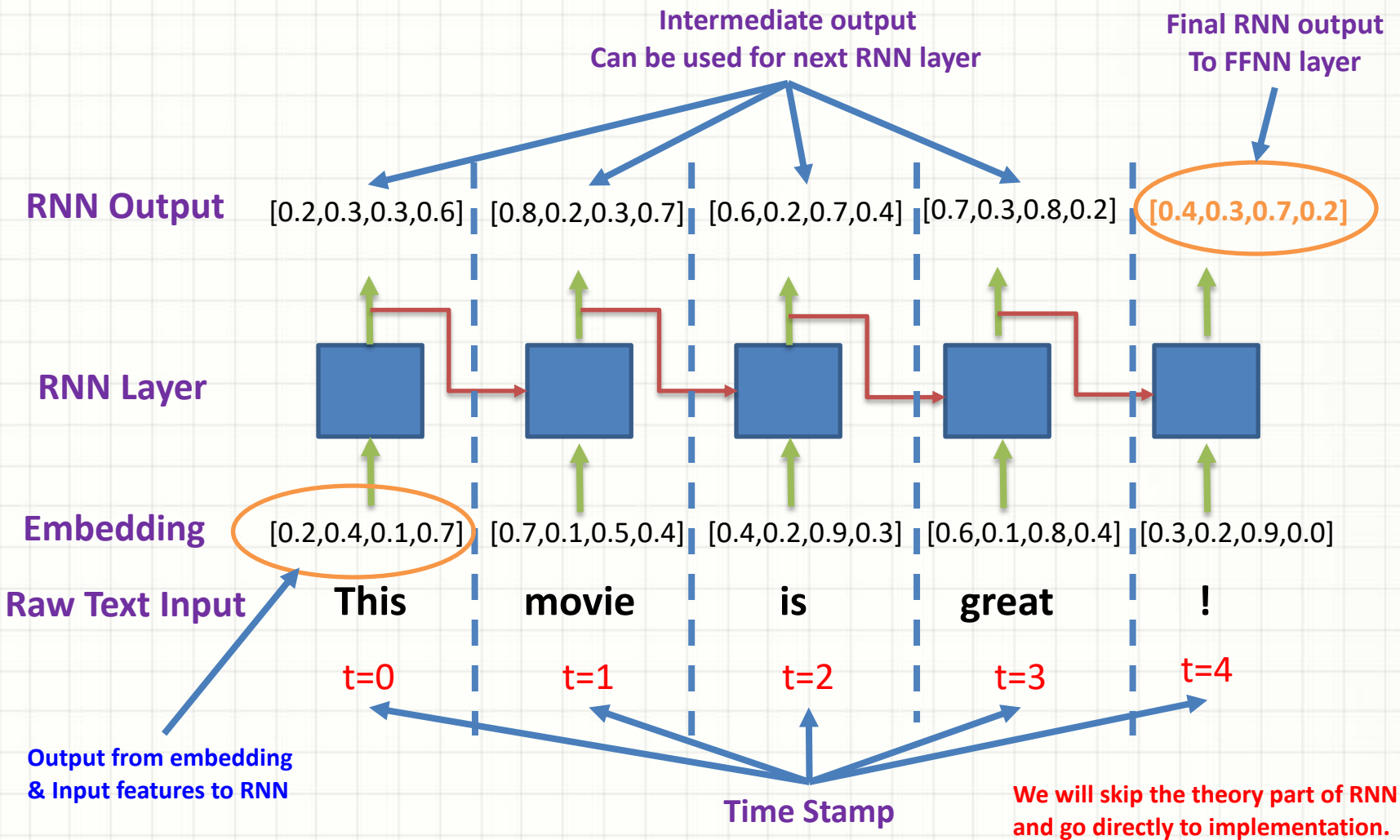
- **word2vec embedding:** word2vec was first introduced in 2013 and it was trained by a large collection of text in an unsupervised fashion. After the training, each word is represented by a fixed length of vector (for example a vector with 300 elements). We can quantify relationships between two words using cosine similarity measure. To train the embedding vector, *continuous bag-of-words* or *continuous skip-gram method* was used. In the continuous bag-of-words architecture, the model predicts the current word from a window of surrounding context words. In the continuous skip-gram architecture, the model uses the current word to predict the surrounding window of context words. There are pretrained word2vec embeddings based on large amount of text (such as wiki pages, news reports, etc) for general applications.
- More detail: https://code.google.com/archive/p/word2vec/

- **fastText embedding:** word2vec uses word-level information, however words are created by meaningful components. fastText use ***subword*** and ***morphological*** information to extend the skip-gram model in word2vec. With fastText, new words not appeared in the training data can be repressed well. It also has 150+ different languages support.
- More detail: https://fasttext.cc/

- **BERT embedding**: Bidirectional Encoder Representations from Transformers which uses contextual information of text in a bi-directional manner.
- More detail: link

Analyzing Text – RNN Modeling

With Embedding (pre-trained or to be trained), we now have a typical data frame for model training: many-to-one RNN structure where the sequence of the word is considered as input and the RNN layer output is connected through a fully-connected layer to the binary output.



We will skip the theory part of RNN and go directly to implementation.

IMDB dataset

- Raw data: 50,000 movie review text (X) and it's corresponding sentiment of positive (1, 50%) or negative (0, 50%) (Y).
- Included in keras R package, can be easily loaded and preprocessed with build-in R functions
- Preprocessing includes:
  - Set size of the vocabulary (i.e. N most frequently occurring words)
  - Set length of the review by padding using '0' by default or truncating as we have to have same length for all reviews for modeling
  - Any words not in the chosen vocabulary replaced by '2' by default
  - Words are indexed by overall frequency in the chosen vocabulary
- Once the dataset is preprocessed, we can apply embedding and then feed the data to RNN layer.

R Code – Data Preprocessing

```
max_features <- 5000
maxlen <- 30
```

```
my_imdb <- dataset_imdb(num_words = max_features)
str(my_imdb)
```

```
x_train <- my_imdb$train$x
y_train <- my_imdb$train$y
x_test  <- my_imdb$test$x
y_test  <- my_imdb$test$y
```

```
x_train <- pad_sequences(x_train, maxlen = maxlen)
x_test <- pad_sequences(x_test, maxlen = maxlen)

str(x_train)
str(y_train)
```

```
 int [1:25000, 1:30] 18 371 47 12 1272 32 665 8 9 13 ...
 int [1:25000] 1 0 0 1 0 0 1 0 1 0 ...
```

R Code – RNN Modeling

```
rnn_model <- keras_model_sequential()
rnn_model %>%
  layer_embedding(input_dim = max_features, output_dim = 32) %>%
  layer_simple_rnn(units = 16, dropout = 0.2, recurrent_dropout = 0.2) %>%
  layer_dense(units = 1, activation = 'sigmoid')
```

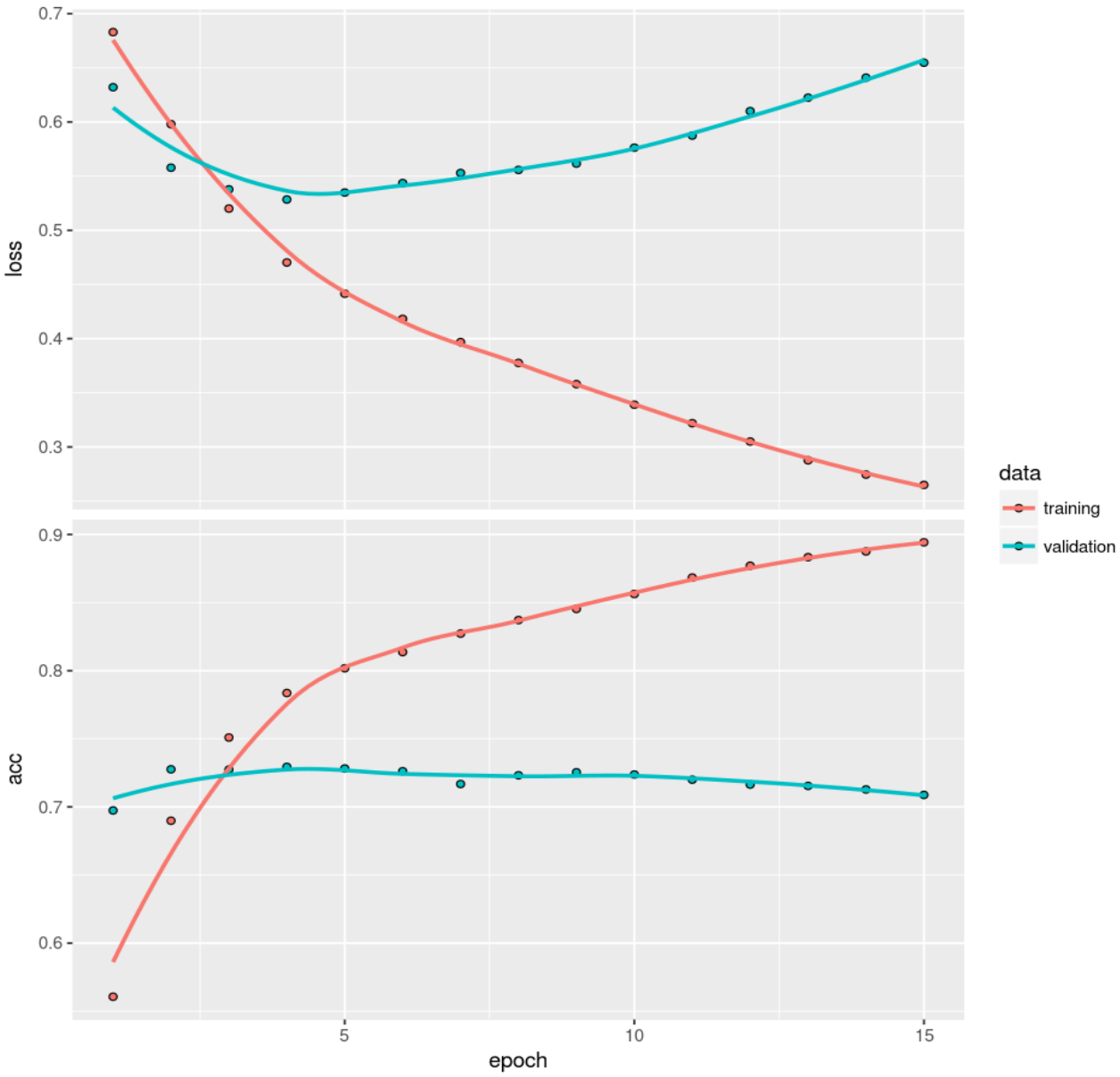
```
rnn_model %>% compile(
  loss = 'binary_crossentropy',
  optimizer = 'adam',
  metrics = c('accuracy')
)
```

```
batch_size = 128
epochs = 15

rnn_history <- rnn_model %>% fit(
  x_train, y_train,
  batch_size = batch_size,
  epochs = 15,
  validation_split = 0.2
)
```

```
rnn_model %>%
   evaluate(x_test, y_test)
```

```
$loss
[1] 0.6454168

$acc
[1] 0.71508
```

```
plot(rnn_history)
```

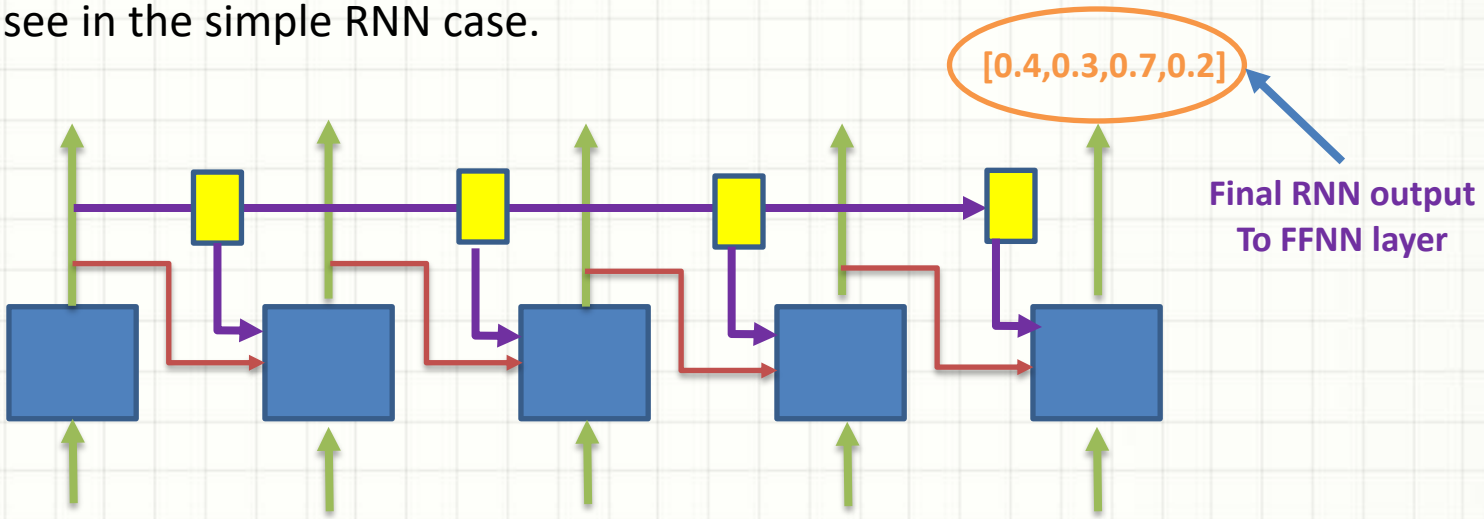


RNN Extension – LSTM

Simple RNN layer is a good starting point, but the performance is usually not that good because long-term dependencies are impossible to learn due to vanishing gradient problem in the optimization process.

LSTM

Long Short Term Memory RNN model introduce a "carry out" mechanism such that useful information from the front words can be carried to later words like a conveyor belt without suffering the vanishing gradient problem we see in the simple RNN case.



A great tutorial about LSTM can be found at:
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Implementation:
layer_simple_rnn() → layer_lstm()

Simple RNN / LSTM Hands-on Session

RNN/LSTM Notebook: link

The End!