

# Popular Machine Learning Methods: Idea, Practice and Math

Part 3, Chapter 1, Section 1:  
Deep Neural Networks

Yuxiao Huang

Data Science, Columbian College of Arts & Sciences  
George Washington University

Fall 2020

# Reference

- This set of slides was largely built on the following 7 wonderful books and a wide range of fabulous papers:
  - HML** Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)
  - PML** Python Machine Learning (3rd Edition)
  - ESL** The Elements of Statistical Learning (2nd Edition)
  - PRML** Pattern Recognition and Machine Learning
  - NND** Neural Network Design (2nd Edition)
  - LFD** Learning From Data
  - RL** Reinforcement Learning: An Introduction (2nd Edition)
- For most materials covered in the slides, we will specify their corresponding books and papers for further reference.

# Code Example

- See related code example in github repository:  
[/p3\\_c2\\_s1\\_deep\\_neural\\_networks/code\\_example](#)

# Table of Contents

- 1 Learning Objectives
- 2 Motivating Example
- 3 Deep Neural Networks
- 4 DNNs
- 5 Building DNNs
- 6 Compiling DNNs
- 7 Training DNNs

# Learning Objectives: Expectation

- It is expected to understand
  - three kinds of the most popular Deep Neural Networks:
    - Fully Connected Feedforward Deep Neural Networks (DNNs)
    - Convolutional Neural Networks (CNNs)
    - Recurrent Neural Networks (RNNs)
  - the good practices for buliding DNNs using tf.keras
  - the good practices for compiling DNNs using tf.keras
  - the good practices for training DNNs using tf.keras
  - the interpretation of:
    - Model Summary
    - History
    - Learning Curve

# Learning Objectives: Recommendation

- It is recommended to understand
  - the relationship between:
    - TensorFlow
    - keras
    - tf.keras

# Fashion MNIST Dataset



Figure 1: Kaggle competition: Fashion MNIST dataset. Picture courtesy of Kaggle.

- Fashion MNIST dataset: a dataset of Zalando's article images:
  - features:  $28 \times 28$  (i.e., 784) pixels (taking value in  $[0, 255]$ ) in a grayscale image
  - target: the article of clothing in each image:
 

● 0: T-shirt/top	● 5: Sandal
● 1: Trouser	● 6: Shirt
● 2: Pullover	● 7: Sneaker
● 3: Dress	● 8: Bag
● 4: Coat	● 9: Ankle boot

# Deep Neural Networks

- In simple words, *Deep Neural Networks* are neural networks with multiple hidden layers.
- Below are three kinds of the most popular deep neural networks:
  - Fully Connected Feedforward Deep Neural Networks (DNNs)
  - Convolutional Neural Networks (CNNs)
  - Recurrent Neural Networks (RNNs)
- Here we will focus on DNNs, which in essence are Multi-Layer Perceptrons (MLPs) with multiple hidden layers.
- We will discuss:
  - CNNs in [/p3\\_c3\\_s3\\_convolutional\\_neural\\_networks](#)
  - RNNs in [/p3\\_c3\\_s4\\_recurrent\\_neural\\_networks](#)



# The Architecture of DNNs

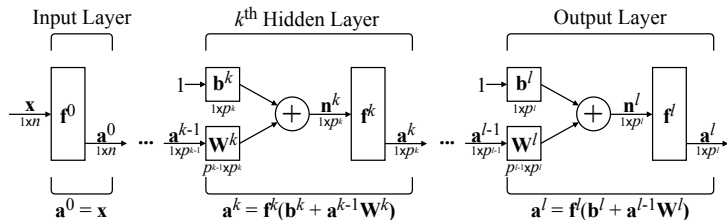


Figure 2: A fully connected feedforward neural network with  $l - 1$  hidden layers.

- $\mathbf{x}$  is a  $1 \times n$  input vector on the input layer (with  $n$  being the number of features).
- $\mathbf{a}^0$  is a  $1 \times n$  output vector on the input layer.
- $\mathbf{a}^{k-1}$  is the  $1 \times p^{k-1}$  input vector on the  $k^{\text{th}}$  hidden layer (with  $p^{k-1}$  being the number of perceptrons on the  $(k-1)^{\text{th}}$  hidden layer).
- $\mathbf{a}^k$  is the  $1 \times p^k$  output vector on the  $k^{\text{th}}$  hidden layer (with  $p^k$  being the number of perceptrons on the  $k^{\text{th}}$  hidden layer).
- $\mathbf{a}^{l-1}$  is the  $1 \times p^{l-1}$  input vector on the output layer (with  $p^{l-1}$  being the number of perceptrons on the last hidden layer).
- $\mathbf{a}^l$  is the  $1 \times p^l$  output vector on the output layer (with  $p^l$  being the number of perceptrons on the output layer).

# The Architecture of DNNs

- Some hyperparameters in the architecture of DNNs are fixed:
  - the number of perceptrons on the input layer ( $p^0$ ) = the number of features in the input ( $n$ )
  - the activation function on the input layer ( $f^0$ ) = identity function:

$$f^0(\mathbf{x}) = \mathbf{x} \quad (1)$$

- Some hyperparameters in the architecture of DNNs are not fixed:
  - the number of hidden layers
  - the number of perceptrons on the hidden layers and output layer ( $p^k$  where  $k > 0$ )
  - the activation function on the hidden layers and output layer ( $f^k$  where  $k > 0$ )
  - the loss function  $\mathcal{L}$

# The Good Practice for Building DNNs

- On pages 12 to 15, we will introduce some good practices for building DNNs.
- Concretely, we will provide recommendations for the architecture of DNNs, which handle the following kinds of problems:
  - Regression
  - Binary classification
  - Multilabel classification
  - Multiclass classification

# Typical Architecture of DNNs for Regression



## Good practice

Hyperparameter	Typical value
# Input layer perceptrons	One per input feature
# Hidden layers	Depends on the problem but typically 1 to 5
# Perceptrons per hidden layer	Depends on the problem but typically 10 to 100
# Output layer perceptrons	1 per prediction dimension
Input layer activation	Linear
Hidden layer activation	ReLU (or SELU)
Output layer activation	Linear or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Table 1: Typical architecture of DNNs for regression.

# Typical Architecture of DNNs for Binary Classification



## Good practice

Hyperparameter	Typical value
# Input layer perceptrons ★	One per input feature
# Hidden layers ★	Depends on the problem but typically 1 to 5
# Perceptrons per hidden layer ★	Depends on the problem but typically 10 to 100
# Output layer perceptrons	1
Input layer activation ★	Linear
Hidden layer activation ★	ReLU (or SELU)
Output layer activation	Logistic
Loss function	Cross entropy

**Table 2:** Typical architecture of DNNs for binary classification (here ★ indicates that the corresponding hyperparameter value is the same as that in table 1).

# Multilabel Classification



## Good practice

Hyperparameter	Typical value
# Input layer perceptrons ★	One per input feature
# Hidden layers ★	Depends on the problem but typically 1 to 5
# Perceptrons per hidden layer ★	Depends on the problem but typically 10 to 100
# Output layer perceptrons	1 per label
Input layer activation ★	Linear
Hidden layer activation ★	ReLU (or SELU)
Output layer activation	Logistic
Objective function	Cross entropy

**Table 3:** Typical architecture of DNNs for multilabel classification (here ★ indicates that the corresponding hyperparameter value is the same as that in table 1).

# Multiclass Classification



## Good practice

Hyperparameter	Typical value
# Input layer perceptrons ★	One per input feature
# Hidden layers ★	Depends on the problem but typically 1 to 5
# Perceptrons per hidden layer ★	Depends on the problem but typically 10 to 100
# Output layer perceptrons	1 per class
Input layer activation ★	Linear
Hidden layer activation ★	ReLU (or SELU)
Output layer activation	Softmax
Objective function	Cross entropy

**Table 4:** Typical architecture of DNNs for multiclass classification (here ★ indicates that the corresponding hyperparameter value is the same as that in table 1).

# The Most Popular Tools for Deep Neural Networks

- Below are the most popular tools for deep neural networks:
  - [TensorFlow](#)
  - [Keras](#)
  - [PyTorch](#)
- In this course we will be using [tf.keras](#), which is a tool with Keras API specification and TensorFlow specific backend.



# Building DNNs

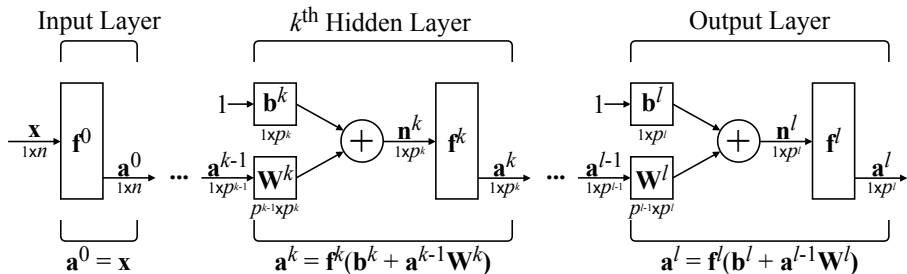


Figure 1: A fully connected feedforward neural network with  $l - 1$  hidden layers.

- The sequential API in `tf.keras` allows us to build DNNs (where the layers are fully connected in a sequential way, as shown in fig. 1).

# Building DNNs: Code Example

- See [/p3\\_c2\\_s1\\_deep\\_neural\\_networks/code\\_example:](#)
  - ① cells 16 and 17

# Model Summary

```
In [18]: model.summary()
```

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
=====		
flatten_1 (Flatten)	(None, 784)	0
dense_3 (Dense)	(None, 50)	39250
dense_4 (Dense)	(None, 50)	2550
dense_5 (Dense)	(None, 10)	510
=====		
Total params: 42,310		
Trainable params: 42,310		
Non-trainable params: 0		

Figure 3: Model summary.

# Interpreting Model Summary

- In column Layer (type): Flatten denotes a flatten layer and Dense denotes a dense (fully connected) layer.
- In column Output Shape: None means the batch size can take any value.
- In column Param: the number of parameters on layer  $k$  is

$$p^{k-1}p^k + p^k. \quad (2)$$

Here:

- $p^{k-1}$  is the number of perceptrons on layer  $k - 1$  (also the number of bias on the layer)
- $p^k$  is the number of perceptrons on layer  $k$  (also the number of bias on the layer)
- $p^k p^{k-1}$  is the number of connecting weights between layer  $k - 1$  and  $k$
- For example:
  - on layer dense\_3:  $39250 = 784 \times 50 + 50$
  - on layer dense\_4:  $2550 = 50 \times 50 + 50$
  - on layer dense\_5:  $510 = 50 \times 10 + 10$
- Trainable params are the parameters that will be updated by backpropagation.
- Non-trainable params are the parameters that will not be updated by backpropagation.

# Compiling DNNs: Code Example

- After building DNNs, we need to compile it to specify hyperparameters such as:
  - loss function
  - optimizer
  - evaluation metrics
- See [/p3\\_c2\\_s1\\_deep\\_neural\\_networks/code\\_example:](#)
  - ① cell 19

# Training DNNs: Code Example

- After compiling DNNs, we can train the model on the training data and evaluate it on the validation data.
- See [/p3\\_c2\\_s1\\_deep\\_neural\\_networks/code\\_example](/p3_c2_s1_deep_neural_networks/code_example):
  - 1 cell 20

# History

```
In [20]: # Train and evaluate the model
history = model.fit(data_train,
                    epochs=10,
                    validation_data=data_valid)
```

Epoch 1/10  
 2625/2625 [=====] - 15s 6ms/step - loss: 0.7334 - accuracy: 0.7470 - val\_loss: 0.5248 - val\_accuracy: 0.8157  
 Epoch 2/10  
 2625/2625 [=====] - 15s 6ms/step - loss: 0.5012 - accuracy: 0.8227 - val\_loss: 0.4748 - val\_accuracy: 0.8344  
 Epoch 3/10  
 2625/2625 [=====] - 15s 6ms/step - loss: 0.4515 - accuracy: 0.8407 - val\_loss: 0.4361 - val\_accuracy: 0.8469  
 Epoch 4/10  
 2625/2625 [=====] - 15s 6ms/step - loss: 0.4231 - accuracy: 0.8510 - val\_loss: 0.4169 - val\_accuracy: 0.8526  
 Epoch 5/10  
 2625/2625 [=====] - 15s 6ms/step - loss: 0.4001 - accuracy: 0.8591 - val\_loss: 0.3936 - val\_accuracy: 0.8580  
 Epoch 6/10  
 2625/2625 [=====] - 15s 6ms/step - loss: 0.3845 - accuracy: 0.8646 - val\_loss: 0.3806 - val\_accuracy: 0.8637  
 Epoch 7/10  
 2625/2625 [=====] - 16s 6ms/step - loss: 0.3712 - accuracy: 0.8680 - val\_loss: 0.3793 - val\_accuracy: 0.8608  
 Epoch 8/10  
 2625/2625 [=====] - 16s 6ms/step - loss: 0.3606 - accuracy: 0.8710 - val\_loss: 0.3734 - val\_accuracy: 0.8663  
 Epoch 9/10  
 2625/2625 [=====] - 16s 6ms/step - loss: 0.3489 - accuracy: 0.8751 - val\_loss: 0.3642 - val\_accuracy: 0.8682  
 Epoch 10/10  
 2625/2625 [=====] - 16s 6ms/step - loss: 0.3410 - accuracy: 0.8774 - val\_loss: 0.3563 - val\_accuracy: 0.8717

Figure 4: The training and validation history across 10 epochs.

# Interpreting History

- In each epoch, the history displays:
  - column 1: the number of sample processed so far, with the corresponding progress bar
  - column 2: the mean training time per sample
  - remaining columns: the metrics (e.g., loss and accuracy) specified when compiling the model, on the training data and validation data (if specified when training the model)
- Fig. 4 shows that, when training progresses (generally):
  - the training and validation loss decrease
  - the training and validation accuracy increase



# The Learning Curve

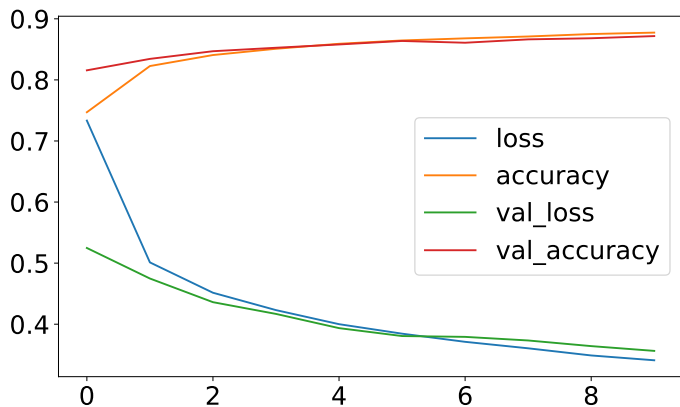


Figure 5: The learning curve of training and validation across 10 epochs.

# Interpreting the Learning Curve

- Fig. 5 shows that, when training progresses (generally):
  - the training and validation loss (the blue and green curve) decrease
  - the training and validation accuracy (the orange and red curve) increase
- A closer look at the figure shows that the validation loss/accuracy is lower/higher than the training loss/accuracy.
- However, this is not the case since the validation metrics are computed at the end of each epoch, whereas the training metrics are computed using a running mean during each epoch.
- In other words, the validation metrics are half an epoch (due to the running mean) ahead of the training metrics.



## Good practice

- When comparing the learning curve on the training and validation data, we should shift the curve on the training data by half an epoch to the left.

# Saving and Loading DNNs

- It took us several minutes to train the four layer DNN (with 50 perceptrons on each hidden layer) on the Fashion MNIST dataset (with 60,000  $28 \times 28$  images).
- In reality, both the DNNs and the dataset can be much more complex, resulting in even higher training time.
- If the server crashed, the memory would be erased.
- As a result, the model, which resides in the memory during training, would be lost.
- For this reason we should periodically save the model on the disk when training processes, so that we can load the saved model from the disk (even when the server crashed).



## Good practice

- We should periodically save the model on the disk when training processes, so that we can load the saved model from the disk (even when the server crashed).

# The Checkpoint Callback: Code Example

- Fortunately, we can use the *Checkpoint Callback* to save DNNs on the disk and load the saved DNNs from the disk.
- See [/p3\\_c2\\_s1\\_deep\\_neural\\_networks/code\\_example](/p3_c2_s1_deep_neural_networks/code_example):
  - 1 cells 28 and 29

# Continuing Training DNNs: Code Example

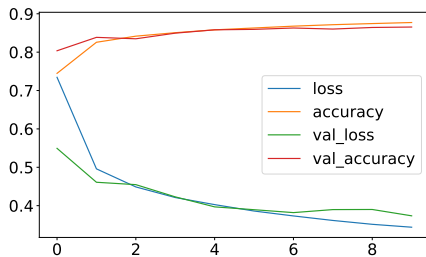


Figure 6: The learning curve of training and validation across 10 epochs.

- Fig. 6 suggest that, the validation metrics would have been better had we trained the model longer.
- Luckily we can simply use the `fit()` function again to continue the training from where we left off.
- See [/p3\\_c2\\_s1\\_deep\\_neural\\_networks/code\\_example:](/p3_c2_s1_deep_neural_networks/code_example:)
  - cells 31 and 32

# History

In [32]: *# Train, evaluate and save the model*

```
history = model.fit(data_train,
                    epochs=5,
                    validation_data=data_valid,
                    callbacks=[model_checkpoint_cb])
```

```
Epoch 1/5
2625/2625 [=====] - 16s 6ms/step - loss: 0.3352 - accuracy: 0.8805 - val_loss: 0.3750 - val_
accuracy: 0.8638
Epoch 2/5
2625/2625 [=====] - 16s 6ms/step - loss: 0.3278 - accuracy: 0.8825 - val_loss: 0.3488 - val_
accuracy: 0.8736
Epoch 3/5
2625/2625 [=====] - 16s 6ms/step - loss: 0.3213 - accuracy: 0.8848 - val_loss: 0.3566 - val_
accuracy: 0.8719
Epoch 4/5
2625/2625 [=====] - 16s 6ms/step - loss: 0.3144 - accuracy: 0.8862 - val_loss: 0.3556 - val_
accuracy: 0.8735
Epoch 5/5
2625/2625 [=====] - 16s 6ms/step - loss: 0.3085 - accuracy: 0.8885 - val_loss: 0.3753 - val_
accuracy: 0.8679
```

Figure 7: The training and validation history across the continued 5 epochs.

# The Learning Curve

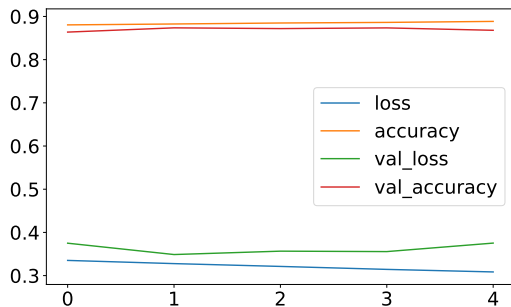


Figure 8: The learning curve of training and validation across the continued 5 epochs.

# Overriding the Best Model

- Both the history (fig. 7) and learning curve (fig. 8) show that, while the metrics on the training data keep getting better, the ones on the validation sometimes get better (from epoch 1 to 2 and epoch 3 to 4) but sometimes get worse (from epoch 2 to 3 and epoch 4 to 5).
- The results above suggest that we could be overfitting after epoch 2 (in other words, the model obtained in epoch 2 could be the best one).
- Unfortunately, the checkpoint callback will override the best model with the one obtained in the last epoch.
- Would not it be great if we only save the best model?



# Checkpoint Callback: Code Example

- Fortunately, we can tweak the checkpoint callback to do so.
- See [/p3\\_c2\\_s1\\_deep\\_neural\\_networks/code\\_example:](#)
  - ① cells 36 and 37

# Tradeoff between Accuracy and Speed

- In terms of accuracy, we would like the epoch number to be as large as possible, since the larger the number the better the model (when using checkpoint callback).
- In terms of speed, we would like the epoch number to be as small as possible, since the smaller the number the faster the training.
- Intuitively we want to have a good tradeoff between accuracy and speed.
- One approach for doing so is *Early Stopping*.
- As discussed in [/p2\\_c2\\_s5\\_tree\\_based\\_models](#), the idea of early stopping is that, we monitor the validation metrics (e.g., loss or accuracy) and terminate the training as soon as the accuracy stops improving for some consecutive epochs.

# EarlyStopping Callback: Code Example

- Fortunately, we can combine the checkpoint and earlystopping callback to strike a good balance between accuracy and speed.
- See [/p3\\_c2\\_s1\\_deep\\_neural\\_networks/code\\_example](/p3_c2_s1_deep_neural_networks/code_example):
  - 1 cells 41 and 42