Popular Machine Learning Methods: Idea, Practice and Math

Part 3, Chapter 2, Section 2: Training Deep Neural Networks

Yuxiao Huang

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

Spring 2021

Reference

- This set of slices 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 & Case Study

- See related code example in github repository: /p3_c2_s2_training_deep_neural_networks/code_example
- See related case study in github repository: /p3_c2_s2_training_deep_neural_networks/case_study

Table of Contents

Learning Objectives

- It is **expected** to understand
  - the idea of Learning Rate Scheduling
  - the idea of the Vanishing / Exploding Gradient Problem
  - the idea of and good practices for Nonsaturating Activation Functions
  - the idea of and good practices for Batch Nomalization
  - the idea of and good practices for Gradient Clipping
  - the idea of and good practices for Early Stopping
  - the idea of and good practices for Dropout
  - the idea and good practices for Transfer Learning

Learning Objectives: Recommendation

- It is **recommended** to understand
  - the math of Learning Rate Scheduling
  - the math of the Vanishing / Exploding Gradient Problem
  - the math of Nonsaturating Activation Functions
  - the math of Batch Nomalization
  - the math of Gradient Clipping
  - the math of Dropout

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: 28×28 (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
    - 1: Trouser
    - 2: Pullover
    - 3: Dress
    - 4: Coat
    - 5: Sandal
    - 6: Shirt
    - 7: Sneaker
    - 8: Bag
    - 9: Ankle boot

CIFAR-10

Figure 2: Kaggle competition: CIFAR-10 dataset. Picture courtesy of Kaggle.

- CIFAR-10 dataset: a dataset for image classification:
  - features: 32×32 (i.e., 1024) pixels (taking value in $[0, 255]$) in a color image
  - target: the object in each image:
    - 0: airplane
    - 1: automobile
    - 2: bird
    - 3: cat
    - 4: deer
    - 5: dog
    - 6: frog
    - 7: horse
    - 8: ship
    - 9: truck

Fine-tuning DNNs

- Fine-tuning DNNs involves fine-tuning:
  - the architecture
  - the hyperparameters
- Architecture includes, for example:
  - number of hidden layers
  - number of perceptrons on each hidden layer
  - activation function on each hidden layer
- Hyperparameters include, for example:
  - learning rate (very important)
  - optimizer
- Here we will focus on fine-tuning the learning rate (as it is the most important hyperparameter).
- See a very nice discussion of faster optimizers (compared to traditional gradient descent optimizer in HML: Chap 11).

Hidden Layers and Perceptrons per Hidden Layer

Good practice

- If there are state-of-the-art DNNs pretrained on similar data:
  - use Transfer Learning (more on this later)
- Otherwise:
  - # hidden layers:
    - depends on the problem
    - typically 1 to 5 (as discussed in /p3_c2_s1_deep_neural_networks)
  - # perceptrons on each hidden layer:
    - usually same on each hidden layer (except for some DNNs, e.g., auto-encoder)
    - typically 10 to 100 (as discussed in /p3_c2_s1_deep_neural_networks)
  - # hidden layers and # perceptrons on each hidden layer:
    1. use larger number of hidden layers and perceptrons
    2. use regularization (e.g., Early Stopping and Dropout, more on this later) to prevent overfitting
    3. this is as known as the *Stretch Pants* approach

Learning Rate

- Instead of fine-tuning the learning rate (using methods such as grid search) to find a constant value for DNNs, it is recommended to use *Learning Rate Scheduling* to adjust learning rate during training.
- Below are some of the most widely used learning rate scheduling methods:
  - Power Scheduling
  - Exponential Scheduling
  - Performance Scheduling

Good practice

- It is recommended to use learning rate scheduling to adjust learning rate during training.

Power Scheduling

- In *Power Scheduling*, learning rate decreases as training progresses.
- Concretely, learning rate is a function of the iteration number:

$$\eta(t) = \frac{\eta_0}{\left(1 + \frac{t}{s}\right)^c}. \tag{1}$$

- Here:
  - $\eta(t)$ is the learning rate at iteration t
  - η_0 is the initial learning rate
  - s is the step size
  - c is the power (typically 1)
- Based on eq. (1), we have the following relationship between learning rate at step ks, $\eta(ks)$, and that at step $(k+1)s$, $\eta\big((k+1)s\big)$:

$$\frac{\eta\big((k+1)s\big)}{\eta(ks)} = \frac{(k+1)^c}{(k+2)^c} = \left(1 - \frac{1}{k+2}\right)^c. \tag{2}$$

- The relationship suggests that the decrease of learning rate gets smaller when training progresses.

Power Scheduling: Code Example

- See /p3_c2_s2_training_deep_neural_networks/code_example:
  1. cell 18

Exponential Scheduling

- Similar to power scheduling, in *Exponential Scheduling* learning rate also decreases as training progresses.
- Concretely, learning rate is a function of the iteration number:

$$\eta(t) = \eta_0 0.1^{\frac{t}{s}}. \tag{3}$$

- Here:
  - $\eta(t)$ is the learning rate at iteration t
  - η_0 is the initial learning rate
  - s is the step size
- Based on eq. (3), we have the following relationship between learning rate at step ks, $\eta(ks)$, and that at step $(k+1)s$, $\eta\big((k+1)s\big)$:

$$\frac{\eta\big((k+1)s\big)}{\eta(ks)} = \frac{0.1^{\frac{k+1}{s}}}{0.1^{\frac{k}{s}}} = 0.1. \tag{4}$$

- The relationship suggests that, unlike power scheduling where the decrease of learning rate gets smaller when training progresses, the decrease in exponential scheduling remains the same (by a factor of 10 every s step).

Exponential Scheduling: Code Example

- See /p3_c2_s2_training_deep_neural_networks/code_example:
  1. cell 24

Performance Scheduling

- In *Performance Scheduling*, learning rate is decreased by a factor of λ as soon as the validation metrics (e.g., validation loss) have not improved for some consecutive steps.

Performance Scheduling: Code Example

- See /p3_c2_s2_training_deep_neural_networks/code_example:
  1. cells 30 and 31

Further Reading

- See HML: Chap 11 for more learning rate scheduling methods.

Backpropagation: Revisit

- As discussed in /p2_c2_s4_shallow_neural_networks, each iteration of backpropagation begins with forward pass, then backward pass, and ends with gradient descent.
- Forward pass
  1. passes the input to the input layer:
  $$\mathbf{a}^0 = \mathbf{f}^0(\mathbf{x}) = \mathbf{x} \tag{5}$$
  2. calculates the output from the first hidden layer up to the output layer:
  $$\mathbf{a}^k = \mathbf{f}^k(\mathbf{n}^k) = \mathbf{f}^k(\begin{bmatrix}1 & \mathbf{a}^{k-1}\end{bmatrix}\boldsymbol{\theta}^k), \quad \text{where} \quad 1 \leq k \leq l \tag{6}$$
- Backward pass
  1. calculates the sensitivity on the output layer:
  $$\mathbf{s}^l = \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \mathbf{n}^l} = -2(\mathbf{y} - \mathbf{a}^l)\dot{\mathbf{f}}^l(\mathbf{n}^l) \tag{7}$$
  2. calculates the sensitivity from the last hidden layer down to the first:
  $$\mathbf{s}^k = \mathbf{s}^{k+1}\left(\mathbf{W}^{k+1}\right)^{\mathsf{T}}\dot{\mathbf{F}}^k(\mathbf{n}^k), \quad \text{where} \quad l-1 \geq k \geq 1 \tag{8}$$
- Gradient descent
  1. updates the parameters on each layer:
  $$\boldsymbol{\theta}^k = \boldsymbol{\theta}^k - \eta\begin{bmatrix}1 & \mathbf{a}^{k-1}\end{bmatrix}^{\mathsf{T}}\mathbf{s}^k \tag{9}$$

The Backward Pass: Revisit

- Let us take a closer look at the backward pass summarized in eq. (8)

$$\mathbf{s}^k = \mathbf{s}^{k+1}\left(\mathbf{W}^{k+1}\right)^{\mathsf{T}}\dot{\mathbf{F}}^k(\mathbf{n}^k), \quad \text{where} \quad l-1 \geq k \geq 1. \tag{8}$$

Here:

- $\mathbf{s}^k$ is the sensitivity on layer k
- $\mathbf{s}^{k+1}$ is the sensitivity on layer $k+1$
- $\mathbf{W}^{k+1}$ is the weight matrix on layer $k+1$
- $\dot{\mathbf{F}}^k(\mathbf{n}^k)$ is a diagonal matrix:

$$\dot{\mathbf{F}}^k(\mathbf{n}^k) = \begin{bmatrix} \dot{f}_1^k(n_1^k) & 0 & \cdots & 0 \\ 0 & \dot{f}_2^k(n_2^k) & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \dot{f}_{p^k}^k(n_{p^k}^k) \end{bmatrix}, \tag{10}$$

where $\dot{f}_i^k(n_i^k)$ is the partial derivative of the activation function of perceptron i on layer k, $f_i^k(n_i^k)$, with respect to the net input of perceptron i on layer k, n_i^k

The Vanishing / Exploding Gradient Problem

- Let us focus on the diagonal matrix, $\dot{\mathbf{F}}^k(\mathbf{n}^k)$, in eq. (8)

$$\mathbf{s}^k = \mathbf{s}^{k+1}\left(\mathbf{W}^{k+1}\right)^\intercal \dot{\mathbf{F}}^k(\mathbf{n}^k), \quad \text{where} \quad l-1 \geq k \geq 1. \tag{8}$$

- If the partial derivatives (i.e., gradients) in $\dot{\mathbf{F}}^k(\mathbf{n}^k)$, $\dot{f}_i^k(n_i^k)$, are very small, the sensitivity on the lower layers could become smaller and smaller when backward pass proceeds down to the lower layers (think about what happens when multiplying say, 0.1, multiple times), a problem called *Vanishing Gradient*.
- If, on the other hand, the partial derivatives (i.e., gradients) in $\dot{\mathbf{F}}^k(\mathbf{n}^k)$, $\dot{f}_i^k(n_i^k)$, are very large, the sensitivity on the lower layers could become larger and larger when backward pass proceeds down to the lower layers (think about what happens when multiplying say, 10, multiple times), a problem called *Exploding Gradient*.
- Based on the gradient descent summarized in eq. (9)

$$\boldsymbol{\theta}^k = \boldsymbol{\theta}^k - \eta \begin{bmatrix} 1 & \mathbf{a}^{k-1} \end{bmatrix}^\intercal \mathbf{s}^k, \tag{9}$$

  - when vanishing gradient happens (where $\mathbf{s}^k$ is very small), the parameters on the lower layers, $\boldsymbol{\theta}^k$, will hardly be updated, in turn, training will not converge
  - when exploding gradient happens (where $\mathbf{s}^k$ is very large), the parameters on the lower layers, $\boldsymbol{\theta}^k$, will become very large, in turn, training will diverge

Handling the Vanishing / Exploding Gradient Problem

- Below are some of the most popular ways to handle the vanishing / exploding problem:
  - Weight Initialization
  - Nonsaturating Activation Functions
  - Batch Normalization
  - Gradient Clipping
- Here we will focus on
  - nonsaturating activation functions
  - batch normalization
  - gradient clipping
- See a very nice discussion of weight initialization in HML: Chap 11.

Some Popular NonSaturating Activation Functions

- Here are some popular activation functions used in DNNs:
  - Rectified Linear Unit (ReLU):

$$ReLU(x) = \begin{cases} x, & \text{if} \quad x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

  - Leaky Rectified Linear Unit (LeakyReLU):

$$LeakyReLU(x) = \begin{cases} x, & \text{if} \quad x \geq 0 \\ \alpha x, & \text{otherwise} \end{cases} \tag{12}$$

  - Exponential Linear Unit (ELU) [Clevert et al., 2015]:

$$ELU(x) = \begin{cases} x, & \text{if} \quad x \geq 0 \\ e^x - 1, & \text{otherwise} \end{cases} \tag{13}$$

  - Scaled Exponential Linear Unit (SELU) [Klambauer et al., 2017]:

$$SeLU(x) = \lambda \begin{cases} x, & \text{if} \quad x \geq 0 \\ \alpha e^x - \alpha, & \text{otherwise} \end{cases} \tag{14}$$

Comparing the Popular Activation Functions

- Below are the activation functions sorted in descending order of preference (in general):
  1. SeLU
  2. ELU
  3. Leaky ReLU
  4. ReLU
  5. tanh
  6. Sigmoid
- However, as many libraries and hardware accelerators provide ReLU-specific optimizations, ReLU could still be the best choice particularly when run time is the priority.

Good practice

- In general:
  - if run time is the priority, ReLU could be the best choice
  - otherwise, SeLU could be the best choice

Batch Normalization: Idea

- While using nonsaturating activation functions (e.g., ReLU or SeLU) can handle the varnishing / exploding gradient problem in the beginning of training, it may not prevent the problem during training.
- *Batch Normalization* (BN) was proposed to address the varnishing / exploding gradient problem in the beginning of and during training.
- The idea of BN is adding an operation before or after the activation function on each hidden layer, which:
  1. standardizes the input
  2. scales and shifts the standardized input

Batch Normalization: Math (Training)

- When training DNNs, for each mini-batch, $\mathbf{mb}_j$, BN performs the following operations:
  1. calculates the mean of input $\mathbf{X}_j$ with respect to mini-batch $\mathbf{mb}_j$, $\boldsymbol{\mu}_j$:
  $$\boldsymbol{\mu}_j = \frac{1}{|\mathbf{mb}_j|} \sum_{i \in \mathbf{mb}^j} \mathbf{x}_j^i \tag{15}$$
  2. calculates the standard deviation of input $\mathbf{X}_j$ with respect to mini-batch $\mathbf{mb}_j$, $\boldsymbol{\sigma}_j$:
  $$\boldsymbol{\sigma}_j^2 = \frac{1}{|\mathbf{mb}_j|} \sum_{i \in \mathbf{mb}^j} \left(\mathbf{x}_j^i - \boldsymbol{\mu}_j\right)^2 \tag{16}$$
  3. calculates the standardized $\mathbf{X}_j$, $\mathbf{Y}_j$:
  $$\mathbf{Y}_j = \frac{\mathbf{X}_j - \boldsymbol{\mu}_j}{\sqrt{\boldsymbol{\sigma}_j^2 + \epsilon}}, \tag{17}$$
  where ϵ is a *Smoothing Term* (typically 10^{-5}) to avoid division by zero
  4. calculates the scaled and shifted $\mathbf{Y}_j$, $\mathbf{Z}_j$:
  $$\mathbf{Z}_j = \boldsymbol{\gamma} \otimes \mathbf{Y}_j + \boldsymbol{\beta}, \tag{18}$$
  where $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are the scale and shift parameter vector, and $\otimes$ the *Hadamard Product* (a.k.a., *Element-wise Product*)

Batch Normalization: Math (Testing)

- As discussed earlier, when training DNNs we can calculate the mean and standard deviation of the input with respect to a mini-batch, using eqs. (15) and (16):

$$\boldsymbol{\mu}_j = \frac{1}{|\mathbf{mb}_j|} \sum_{i \in \mathbf{mb}^j} \mathbf{x}_j^i, \quad \boldsymbol{\sigma}_j^2 = \frac{1}{|\mathbf{mb}_j|} \sum_{i \in \mathbf{mb}^j} \left(\mathbf{x}_j^i - \boldsymbol{\mu}_j\right)^2. \tag{15, 16}$$

- However, we cannot do so during testing, as the size of test samples could be too small to be useful (in terms of reliably calculating the mean and standard deviation).
- One way to handle this problem is calculating the *Exponential Moving Average* of the mean and standard deviation across all the mini-batches, $\widehat{\boldsymbol{\mu}}$ and $\widehat{\boldsymbol{\sigma}}$:

$$\widehat{\boldsymbol{\mu}}_j = \begin{cases} \boldsymbol{\mu}_1, & \text{if} \quad j = 1, \\ \alpha \boldsymbol{\mu}_j + (1-\alpha)\widehat{\boldsymbol{\mu}}_{j-1}, & \text{if} \quad j > 1. \end{cases} \tag{19}$$

$$\widehat{\boldsymbol{\sigma}}_j = \begin{cases} \boldsymbol{\sigma}_1, & \text{if} \quad j = 1, \\ \alpha \boldsymbol{\sigma}_j + (1-\alpha)\widehat{\boldsymbol{\sigma}}_{j-1}, & \text{if} \quad j > 1. \end{cases} \tag{20}$$

Here:

- $\boldsymbol{\mu}_j$ and $\boldsymbol{\sigma}_j$ are given in eqs. (15) and (16)
- α (where $0 \leq \alpha \leq 1$) is the degree of weighting decrease

Batch Normalization: Pros and Cons

- Pros:
  - significantly alleviate the vanishing gradient problem
  - make DNNs much less sensitive to weight initialization (so that weight initialization may not be necessary)
  - allow training DNNs with much larger learning rates (so that training DNNs can be much faster)
  - can act as a regularizer (so that it can address overfitting)
- Cons:
  - add extra computational cost when:
    - training DNNs (as it needs to standardize, scale, shift the input, and update the parameters)
    - using DNNs for testing (as it needs to standardize, scale and shift the input)

Batch Normalization: Code Example

- See /p3_c2_s1_deep_neural_networks/code_example:
  1. cell 33

Gradient Clipping

- Another method that has been widely used to handle the exploding gradient problem is *Gradient Clipping*.
- The idea of gradient clipping is fairly straightforward: it clips the gradient during backpropagation so that it stays in a predetermined range (hence the name).
- In tf.keras, there are two ways to use gradient clipping:
  - `clipvalue`: clip the gradient so that its absolute value will not be larger than a predetermined threshold
    - `clipvalue` $= 1$ will transform $[0.9, 100]$ into $[0.9, 1.0]$ (where the direction of the gradient is changed)
  - `clipnorm`: clip the gradient so that its l_2 norm will not be larger than a predetermined threshold
    - `clipnorm` $= 1$ will transform $[0.9, 100]$ into $[0.00899964, 0.9999595]$ (where while the direction of the gradient is not changed, the first item in the gradient vector is almost 0)
- We may have to try both `clipvalue` and `clipnorm` and, on top of that, fine-tune the thresholds to better handle the exploding gradient problem.

Gradient Clipping: Code Example

- See /p3_c2_s1_deep_neural_networks/code_example:
  1. cell 39

Handling Overfitting

- Due to the nature of DNNs (a stacking of multiple layers of perceptrons), they tend to result in overfitting.
- Here we will introduce two popular methods to handle the overfitting problem of DNNs:
  - Early Stopping
  - Drop Out

Early Stopping

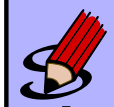
- As discussed in /p2_c2_s5_tree_based_models, the idea of early stopping is that, we monitor the validation metrics (e.g., loss or accuracy) and terminate the training as soon as the accuracy stops improving for some consecutive epochs.
- As discussed in /p3_c2_s1_deep_neural_networks, we can use earlystopping callback to do so.

EarlyStopping Callback: Code Example

- See /p3_c2_s1_deep_neural_networks/code_example:
  1. cells 40 and 41

Dropout

- One of the most popular regularization method for DNNs is called *Dropout*.
- The idea of dropout is fairly simple: for each mini-batch during training, every perceptron on the input layer or a hidden layer will be ignored with probability p.
- Concretely, these ignored perceptrons (by setting its input as 0) will not be used in any step in backpropagation (i.e., forward pass, backward pass or gradient descent).
- The probability p is called *Dropout Rate*, which is typically between 10% and 50%:
  - for CNNs (see /p3_c2_s3_convolutional_neural_networks): about 40%–50%
  - for RNNs (see /p3_c2_s4_recurrent_neural_networks): about 20%–30%



Takeaway

- We use dropout only during training.
- We use dropout only for perceptrons on the input layer and hidden layers.
- We do not use dropout for perceptrons on the output layer.

Good practice

- While in theory we can use dropout for perceptrons on any layer (except for the output layer), in practice we usually use dropout only for perceptrons on the last three to the last one layer (except for the output layer).

Dropout

- Assume the dropout rate, p, is 0.5.
- Since we only use dropout during training, we will be using roughly twice as many perceptrons during testing as it would be during training.
- We have to address this discrepancy, as the net input would roughly twice as large during testing as it would be during training (hence the DNN will not generalize well in reality).
- Generally, there are two ways to handle this problem:
  - after training (but before testing), we multiply the weight parameters on the hidden layers by the *Keep Probability*, $1 - p$
  - during training, we divide the weight parameters on the hidden layers by the *Keep Probability*, $1 - p$
- tf.keras uses the second way to handle the discrepancy between training and testing produced by dropout.

Dropout: Why It Works?

- Since for each mini-batch during training, every perceptron on the input layer or a hidden layer will be ignored with probability p, different mini-batches usually lead to different DNNs.
- As a result, dropout will produce an ensemble of predictors which are fairly similar to bagging (although the predictors produced by dropout are not necessarily independent).
- This is one of the key reasons why dropout, similar to bagging, can handle overfitting.

Dropout: Code Example

- See /p3_c2_s1_deep_neural_networks/code_example:
  1. cell 43

Further Reading

- See HML: Chap 11 for a very nice discussion of *Monte Carlo Dropout*, a method that:
  - takes as input a model trained using dropout and the test data
  - uses the dropout model on the test data multiple times while keeping dropout turning on
  - outputs the average test results

Transfer Learning

- While in theory we can implement DNNs from scratch (as what we did earlier), in reality we are not recommended to do so, particularly when there are state-of-the-art DNNs pretrained on similar data.
- Instead, it is recommended to tweak the pretrained DNNs to make it suitable for our data.
- This approach is called *Transfer Learning*.
- The idea is that:
  - the lower layers of a DNN capture the simple features in the data
  - if our data is similar to the data where a state-of-the-art model was pretrained, then the lower layers of the pretrained model should also be able to capture the simple features in our data
- Transfer learning will not only speed up building, training and fine-tuning DNNs considerably, but also require significantly less training data.

Good practice

- Instead of implementing DNNs from scratch, it is recommended to use transfer learning by tweaking state-of-the-art DNNs pretrained on similar data.
- Transfer learning will not only speed up building, training and fine-tuning DNNs considerably, but also require significantly less training data.

Building DNN with Pretrained Model

Good practice

- To build a DNN with pretrained model, we should:
  1. reuse the lower layers of the pretrained model as the base
  2. add extra layers (that work for our data) on top of the base
- The more similar our data is to the data where the model was pretrained, the more lower layers of the pretrained model we should reuse as the base.
- It is even possible to reuse all the hidden layers of a pretrained model, when the data are similar enough.
- We should resize our data so that the number of features in the resized data is the same as the number of perceptrons on the input layer of the pretrained model.

Training DNN with Pretrained Model

Good practice

- To train a DNN with pretrained model, we should:

1. freeze all the reused layers of the pretrained DNN (i.e., make their weights non-trainable so that backpropagation will not change them) then train the DNN
2. unfreeze one or two top hidden layers of the pretrained DNN (the more training data we have the more top hidden layers we can unfreeze) and reduce the learning rate when doing so (thus the fine-tuned weights on the lower layers will not change significantly)

- If the above steps do not produce an accurate DNN:
  - if we do not have sufficient data, we can drop the top hidden layers and repeat the above steps
  - otherwise, we can replace (rather than drop) the top hidden layers or even add more hidden layers

Building and Training DNN with Pretrained Model

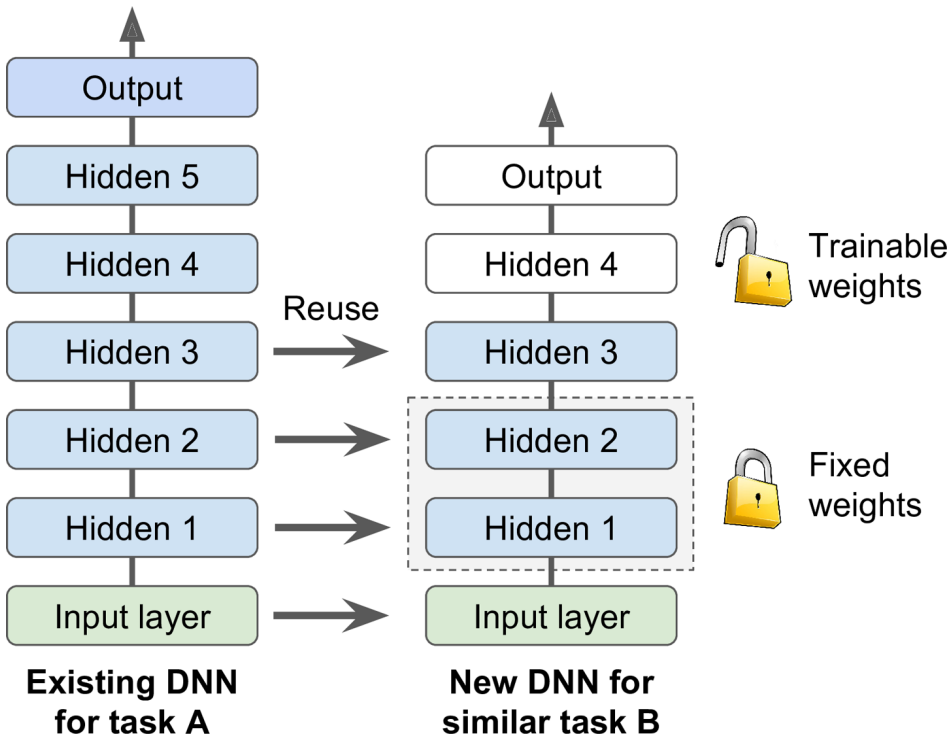


Figure 3: Building and training DNN with pretrained model. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition).*

- Fig. 3 shows the good practices for building and training DNN with pretrained model discussed on pages 41 and 42.

Bibliography I


Clevert, D. A., Unterthiner, T., and Hochreiter, S. (2015).
Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs).
arXiv preprint arXiv:1511.07289.


Klambauer, G., Unterthiner, T., Mayr, A., and Hochreiter, S. (2017).
Self-Normalizing Neural Networks.
In *NeurIPS*, pages 971–980.