Machine Learning I (DATS 6202)
Deep Neural Network

Yuxiao Huang

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

Spring 2020

Reference

- This set of slices was largely built on the following 8 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
LFD	Learning From Data
NND	Neural Network Design (2nd Edition)
NNDL	Neural Network and Deep Learning
RL	Reinforcement Learning: An Introduction

- For most materials covered in the slides, we will specify their corresponding books and papers for further reference.

Code Example & Case Study

- See code example of topics discussed in this section in github repository: /p2_c2_s1_deep_neural_network/code_example
- See case study of Kaggle Competition related to this section in github repository: /p2_c2_s1_deep_neural_network/case_study

Overview

Learning Objectives

- It is **expected** to understand
  - the good practices for designing deep neural network
  - the good practices for training deep neural network
  - the good practices for fine-tuning deep neural network
  - the good practices for transfer learning

Deep Neural Network (DNN)

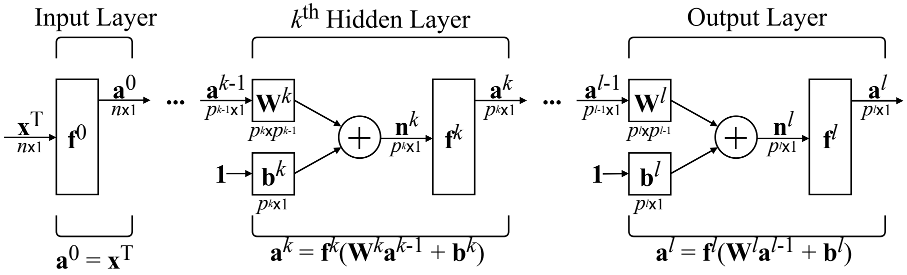


Figure 1: A fully connected neural network.

The Architecture of DNN

- Some hyperparameters in the architecture of DNN 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 ($\mathbf{f}^0$) $=$ linear
- Some hyperparameters in the architecture of DNN 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 functions on the hidden layers and output layer ($\mathbf{f}^k$ where $k > 0$)
  - the objective function J

Regression

- **Good Practice**: Typical hyperparameter value in DNN regressors.

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)
Objective function	MSE or MAE/Huber (if outliers)

Table 1: Typical hyperparameter value in DNN regressors.

Binary Classification

- **Good Practice**: Typical hyperparameter value in DNN binary classifiers. Here $\star$ indicates that the corresponding hyperparameter value is the same as that in DNN regressor (table 1).

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

Table 2: Typical hyperparameter value in DNN binary classifiers.

Multilabel Binary Classification

- **Good Practice**: Typical hyperparameter value in DNN multilabel binary classifiers. Here $\star$ indicates that the corresponding hyperparameter value is the same as that in DNN regressor (table 1).

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

Table 3: Typical hyperparameter value in DNN multilabel binary classifiers.

Multiclass Classification

- **Good Practice**: Typical hyperparameter value in DNN multiclass classifiers. Here $\star$ indicates that the corresponding hyperparameter value is the same as that in DNN regressor (table 1).

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

Table 4: Typical hyperparameter value in DNN multiclass classifiers.

The Most Popular Tools

- Below are the most popular tools for implementing DNN:
  - TensorFlow
  - Keras
  - PyTorch
- In this course we will be using tf.keras.

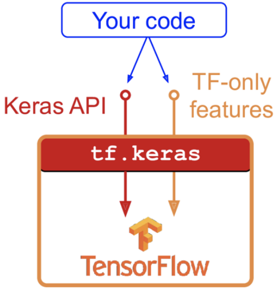


Figure 2: The tf.keras API. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

The Sequential API

- The sequential API allows us to create a DNN where the layers are fully connected in a sequential way (as shown in fig. 3).

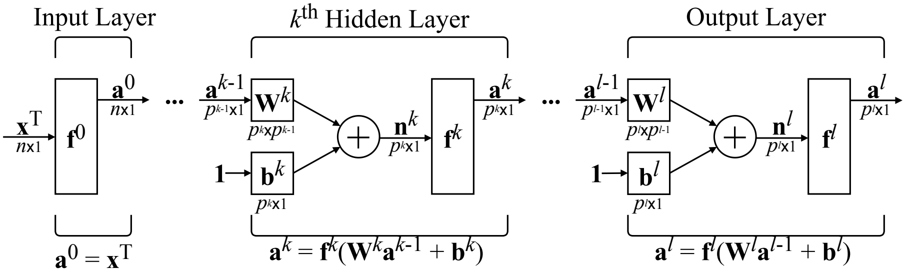


Figure 3: A fully connected neural network.

The Sequential API

- The code below shows one way (by adding the layers one by one) to use the sequential API to create a DNN for the Fashion MNIST dataset (with 60,000 28 $\times$ 28 images):

```
 1  # A sequential dnn
 2  model = keras.models.Sequential()
 3
 4  # Add the input layer
 5  model.add(keras.layers.Flatten(input_shape=[28, 28]))
 6
 7  # Add two hidden layers
 8  model.add(keras.layers.Dense(50, activation="relu"))
 9  model.add(keras.layers.Dense(50, activation="relu"))
10
11  # Add the output layer
12  model.add(keras.layers.Dense(10, activation="softmax"))
```

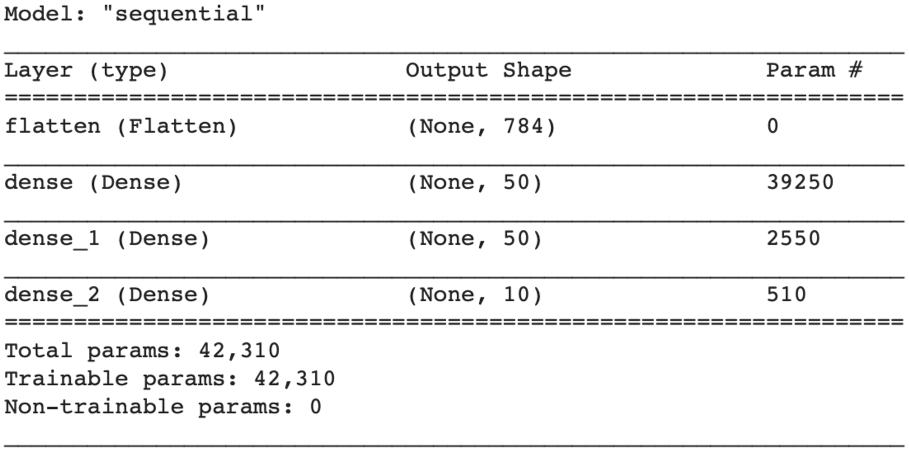
- Line 5: It is recommended to specify the *input_shape* when adding the input layer to a Sequential model.
- Line 5: The flatten layer transforms the input data into a 1d array.

The Sequential API

- The code below shows another way (by passing the layers as a list) to use the sequential API to create a DNN:

```python
# A sequential dnn with two hidden layers
model = keras.models.Sequential([
        keras.layers.Flatten(input_shape=[28, 28]),
        keras.layers.Dense(50, activation="relu"),
        keras.layers.Dense(50, activation="relu"),
        keras.layers.Dense(10, activation="softmax")])
```

Model Summary



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 50)                39250
_________________________________________________________________
dense_1 (Dense)              (None, 50)                2550
_________________________________________________________________
dense_2 (Dense)              (None, 10)                510
=================================================================
Total params: 42,310
Trainable params: 42,310
Non-trainable params: 0
_________________________________________________________________
```

Figure 4: A 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

$$p^k p^{k-1} + p^k, \tag{1}$$

  where p^k is the number of perceptrons on layer k (also the number of bias on the layer), $p^k p^{k-1}$ the number of connecting weights between layer $k-1$ and k. For example:
  - on layer dense: $39250 = 50 \times 784 + 50$
  - on layer dense_1: $2550 = 50 \times 50 + 50$
  - on layer dense_2: $510 = 10 \times 50 + 10$
- Training/Non-trainable params are the parameters that will/will not be updated by backpropagation.

Compiling DNN

- After creating a DNN, we need to compile it to specify hyperparameters such as loss function, optimizer and evaluation metrics.
- The code below shows how to compile a DNN using the *compile()* function:

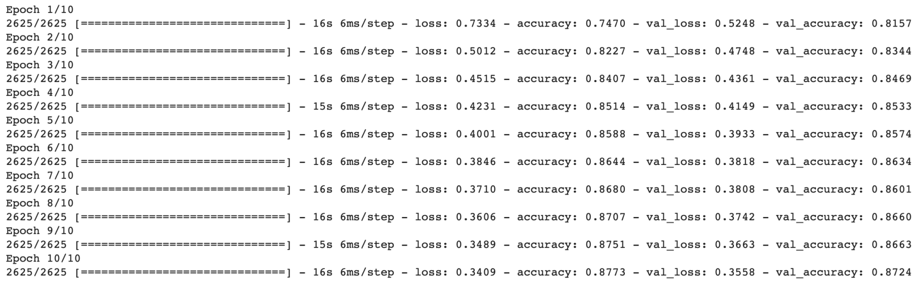
```
# Compile the model
model.compile(optimizer=keras.optimizers.SGD(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Training DNN

- After compiling a DNN, we can train the model on the training data and evaluate it on the validation data.
- The code below shows how to train and evaluate a DNN using the *fit()* function:

```
# Train and evaluate the model
history = model.fit(data_train,
                    epochs=10,
                    validation_data=data_valid)
```

History



```
Epoch 1/10
2625/2625 [==============================] - 16s 6ms/step - loss: 0.7334 - accuracy: 0.7470 - val_loss: 0.5248 - val_accuracy: 0.8157
Epoch 2/10
2625/2625 [==============================] - 16s 6ms/step - loss: 0.5012 - accuracy: 0.8227 - val_loss: 0.4748 - val_accuracy: 0.8344
Epoch 3/10
2625/2625 [==============================] - 16s 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: 0.8514 - val_loss: 0.4149 - val_accuracy: 0.8533
Epoch 5/10
2625/2625 [==============================] - 16s 6ms/step - loss: 0.4001 - accuracy: 0.8588 - val_loss: 0.3933 - val_accuracy: 0.8574
Epoch 6/10
2625/2625 [==============================] - 16s 6ms/step - loss: 0.3846 - accuracy: 0.8644 - val_loss: 0.3818 - val_accuracy: 0.8634
Epoch 7/10
2625/2625 [==============================] - 16s 6ms/step - loss: 0.3710 - accuracy: 0.8680 - val_loss: 0.3808 - val_accuracy: 0.8601
Epoch 8/10
2625/2625 [==============================] - 16s 6ms/step - loss: 0.3606 - accuracy: 0.8707 - val_loss: 0.3742 - val_accuracy: 0.8660
Epoch 9/10
2625/2625 [==============================] - 15s 6ms/step - loss: 0.3489 - accuracy: 0.8751 - val_loss: 0.3663 - val_accuracy: 0.8663
Epoch 10/10
2625/2625 [==============================] - 16s 6ms/step - loss: 0.3409 - accuracy: 0.8773 - val_loss: 0.3558 - val_accuracy: 0.8724
```

Figure 5: The training and validation history across 10 epochs.

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)

The Learning Curve

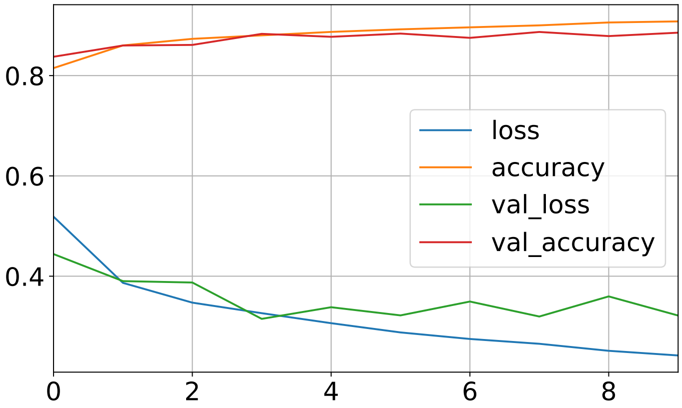


Figure 6: The learning curve of training and validation across 10 epochs.

The Learning Curve

- Fig. 6 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 DNN

- It took us several minutes to train the four layer sequential DNN (with 50 perceptrons on each hidden layer) on the Fashion MNIST dataset (with 60,000 28×28 images).
- In reality, both the DNN and the dataset can be much more complex, resulting in even higher training time.
- If the server crushed, 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. We can then load the saved model from disk.

The Checkpoint Callback

- The code below shows how to save a DNN using the checkpoint callback on the disk and load the saved DNN from disk.

```
 1  # ModelCheckpoint callback
 2  model_checkpoint_cb = keras.callbacks.ModelCheckpoint(
 3      abspath + "/model/model.h5")
 4
 5  # Train, evaluate and save the model
 6  history = model.fit(data_train,
 7                      epochs=10,
 8                      validation_data=data_valid,
 9                      callbacks=[model_checkpoint_cb])
10
11  # Load the saved model
12  model = keras.models.load_model(abspath + "/model/model.h5")
```

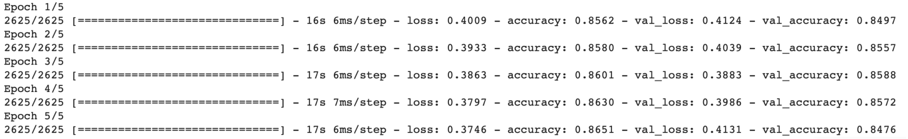
- Line 2 and 12: *abspath* is the absolute path of the working directory.

Continuing Training DNN

- Both fig. 5 and 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.
- The code below shows how to do so.

```
# Train and save the model after each epoch
history = model.fit(data_train,
                    epochs=5,
                    validation_data=data_valid,
                    callbacks=[checkpoint_cb])
```

History



```
Epoch 1/5
2625/2625 [==============================] - 16s 6ms/step - loss: 0.4009 - accuracy: 0.8562 - val_loss: 0.4124 - val_accuracy: 0.8497
Epoch 2/5
2625/2625 [==============================] - 16s 6ms/step - loss: 0.3933 - accuracy: 0.8580 - val_loss: 0.4039 - val_accuracy: 0.8557
Epoch 3/5
2625/2625 [==============================] - 17s 6ms/step - loss: 0.3863 - accuracy: 0.8601 - val_loss: 0.3883 - val_accuracy: 0.8588
Epoch 4/5
2625/2625 [==============================] - 17s 7ms/step - loss: 0.3797 - accuracy: 0.8630 - val_loss: 0.3986 - val_accuracy: 0.8572
Epoch 5/5
2625/2625 [==============================] - 17s 6ms/step - loss: 0.3746 - accuracy: 0.8651 - val_loss: 0.4131 - val_accuracy: 0.8476
```

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

The Learning Curve

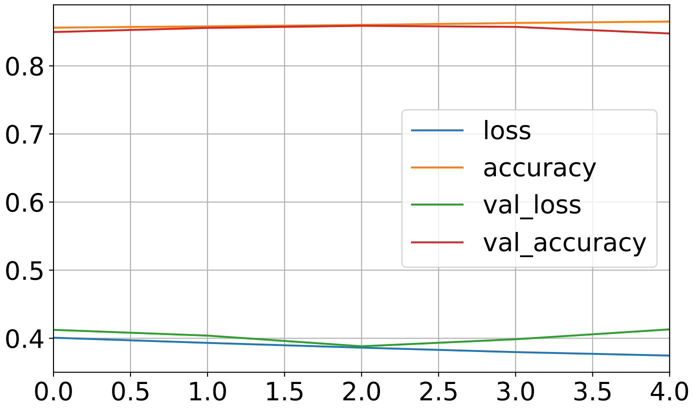


Figure 8: The learning curve of training and validation across the continued 5 epochs.

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 first get better (from epoch 1 to 3) then get worse (from epoch 3 to 5).
- The variance above suggests that we could be overfitting the data after epoch 3. In other words, the model obtained in epoch 3 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

- Fortunately, we can tweak the checkpoint callback to do so:

```
 1 # ModelCheckpoint callback
 2 model_checkpoint_cb = keras.callbacks.ModelCheckpoint(
 3     abspath + "/model/model.h5",
 4     save_best_only=True)
 5 
 6 # Train, evaluate and save the best model
 7 history = model.fit(data_train,
 8                     epochs=20,
 9                     validation_data=data_valid,
10                     callbacks=[model_checkpoint_cb])
```

- Line 4: *save_best_only=True* allows us to only save the best model.
- Line 8: As a result, we can use a large epoch number without worrying about training the model too short (underfitting) or too long (overfitting).

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*.
- The idea 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

- The code below shows how to combine the checkpoint and earlystopping callback to strike a good balance between accuracy and speed:

```
1  # ModelCheckpoint callback
2  model_checkpoint_cb = keras.callbacks.ModelCheckpoint(
3      abspath + "/model/model.h5",
4      save_best_only=True)
5  # EarlyStopping callback
6  early_stopping_cb = keras.callbacks.EarlyStopping(
7      patience=5,
8      restore_best_weights=True)
9
10 # Train, evaluate and save the best model
11 history = model.fit(
12     data_train,
13     epochs=20,
14     validation_data=data_valid,
15     callbacks=[model_checkpoint_cb, early_stopping_cb])
```

- Line 6: By default, EarlyStopping monitors validation loss.
- Line 7: *patience* is the number of consecutive epochs used in EarlyStopping.

Architecture and Hyperparameters

- Fine-tuning DNN involves fine-tuning both the architecture and the hyperparameters.
- Architecture:
  - number of hidden layers and perceptrons per hidden layer
  - activation function of hidden layers
- Hyperparameters:
  - learning rate (very important)
  - optimizer
  - batch size

# Hidden Layers and Perceptrons per Hidden Layer

- **Good Practice**: If there are SOTA DNNs pretrained on similar data:
  - use *Transfer Learning* (more on this later)
- **Good Practice**: Otherwise:
  - # hidden layers: typically 1 to 5 (see tables 1 to 4)
  - # perceptrons per hidden layer:
    - usually same for each hidden layer (except for some DNNs, e.g., auto-encoder)
    - typically 10 to 100 (see tables 1 to 4)
  - # hidden layers and perceptrons per hidden layer:
    1. use larger number of hidden layers and perceptrons
    2. use regularization (e.g., early stopping and drop-out, more on this later) to prevent overfitting
    3. this is as known as the “stretch pants” approach

# Activation Function of Hidden Layers

- **Good Practice**:
  - typically ReLU (or SELU, see tables 1 to 4)
  - not essential to fine-tune

Learning Rate

- **Good Practice**: Instead of using hyperparameter tuning to find a constant learning rate, it is better to use *Learning Scheduling* to adjust learning rate during training.
- Here are some most widely used learning scheduling approaches:
    - Power scheduling
    - Exponential scheduling
    - Performance scheduling

Power Scheduling

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

$$\eta(t) = \frac{\eta_0}{\left(1 + \frac{t}{s}\right)^c}. \tag{2}$$

- 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. (2), we have the following relationship between learning rate at step ks, $\eta(ks)$, and that at step $(k+1)s$, $\eta\big((k+1)s\big)$:

$$\frac{\eta\big((k+1)s\big)}{\eta(ks)} = \frac{(k+1)^c}{(k+2)^c} = \left(1 - \frac{1}{k+2}\right)^c. \tag{3}$$

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

Power Scheduling

- The code below shows how to use power scheduling by setting the *decay* hyperparameter of an optimizer:

```
1 # Compile the model
2 model.compile(optimizer=keras.optimizers.SGD(decay=0.001),
3               loss='sparse_categorical_crossentropy',
4               metrics=['accuracy'])
```

- Line 2: *decay* is the multiplicative inverse of the step size (s) in power scheduling.
- Line 2: The power (c) in power scheduling is set to 1.

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:

$$\eta(t) = \eta_0 0.1^{\frac{t}{s}}. \tag{4}$$

- 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. (4), we have the following relationship between learning rate at step ks, $\eta(ks)$, and that at step $(k+1)s$, $\eta\big((k+1)s\big)$:

$$\frac{\eta\big((k+1)s\big)}{\eta(ks)} = \frac{0.1^{\frac{k+1}{s}}}{0.1^{\frac{k}{s}}} = 0.1. \tag{5}$$

- 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

- The code below shows how to use exponential scheduling by setting the *learning_rate* hyperparameter of an optimizer:

```
 1  # Get the number of samples in the training data
 2  m = tf.data.experimental.cardinality(data_train).numpy()
 3
 4  # Set the decay_steps
 5  s = 20 * m / batch_size
 6
 7  # Get the learning rate
 8  learning_rate = keras.optimizers.schedules.ExponentialDecay(
 9      initial_learning_rate=0.01,
10      decay_steps=s,
11      decay_rate=0.1)
12
13  # Compile the model
14  model.compile(optimizer=keras.optimizers.SGD(learning_rate),
15                loss='sparse_categorical_crossentropy',
16                metrics=['accuracy'])
```

- Line 8: The *learning_rate* is set by *keras.optimizers.schedules.ExponentialDecay*.

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

- The code below shows how to use exponential scheduling by setting the *learning_rate* hyperparameter of an optimizer:

```
 1  # ReduceLROnPlateau callback
 2  reduce_lr_on_plateau_cb = keras.callbacks.ReduceLROnPlateau(
 3      factor=0.1,
 4      patience=5)
 5
 6  # Train, evaluate and save the best model
 7  history = model.fit(
 8      data_train,
 9      epochs=20,
10      validation_data=data_valid,
11      callbacks=[model_checkpoint_cb,
12                 early_stopping_cb,
13                 reduce_lr_on_plateau_cb])
```

- Line 3: *factor* is the factor by which we decrease the learning rate.
- Line 4: *patience* is the number of consecutive steps for which the validation metrics stop improving.

Further Reading

- See HML Chapter 11 for more learning scheduling approaches.

Transfer Learning

- While in theory we can implement our own DNN from scratch (as what we did earlier), in reality we are not recommended to do so, when there are SOTA DNNs pretrained on similar data.
- Instead, we are suggested to tweak the pretrained DNN 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 the pretrained model come from, then the lower layers of the pretrained model should also be able to capture the features in our data.
- Transfer learning will not only speed up designing, training and fine-tuning the DNN considerably, but also require significantly less training data.

Designing DNN with Pretrained DNN

- **Good Practice**:
  - to design 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 the data are, the more lower layers of a pretrained DNN we should reuse
  - it is even possible to reuse all the hidden layers of a pretrained DNN, 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 DNN

Training DNN with Pretrained DNN

- **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

Designing and Training DNN with Pretrained DNN

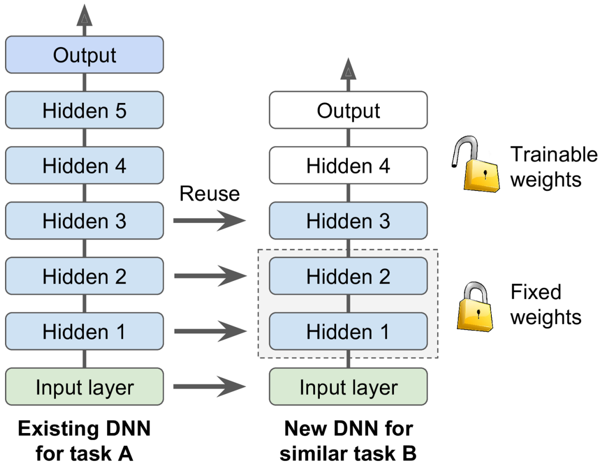


Figure 9: Designing and Training DNN with Pretrained DNN.