

Day 76 Optimizers

優化器 Optimizers 簡介



出題教練

陳宇春

知識地圖 深度學習組成概念

優化器

深度神經網路 Supervised Learning Deep Neural Network (DNN)

- 簡介 Introduction
- 套件介紹 Tools: Keras
- 組成概念 Concept
- 訓練技巧 Training Skill
- 應用案例 Application

卷積神經網路 Convolutional Neural Network (CNN)

- 簡介 introduction
- 套件練習 Practice with Keras
- 訓練技巧 Training Skill
- 電腦視覺 Computer Vision

深度學習組成概念 Concept of DNN

感知器概念簡介



本日知識點目標

- 了解 優化器 (optimizers) 的使用與程式樣貌
- 理解 優化器 (optimizers) 的用途
- 能從程式中辨識 優化器 (optimizers) 的參數特徵

什麼是優化算法 - Optimizer

- 機器學習算法當中，大部分算法的本質就是建立優化模型，通過最優化方法對目標函數進行優化從而訓練出最好的模型
- 優化算法的功能，是通過改善訓練方式，來最小化(或最大化)損失函數 $E(x)$
- 優化策略和算法，是用來更新和計算影響模型訓練和模型輸出的網絡參數，使其逼近或達到最優值

最常用的優化算法-Gradient Descent

- 最常用的優化算法是梯度下降
 - 這種算法使用各參數的梯度值來最小化或最大化損失函數 $E(x)$ 。
- 通過尋找最小值，控制方差，更新模型參數，最終使模型收斂
- 複習一下，前兩堂提到的 Gradient Descent

$$w_{i+1} = w_i - d_i \cdot \eta_i, \quad i=0,1,\dots$$

- 參數 η 是學習率。這個參數既可以設置為固定值，也可以用一維優化方法沿著訓練的方向逐步更新計算
- 參數的更新分為兩步：第一步計算梯度下降的方向，第二步計算合適的學習

複習 – 動量Momentum

- 「一顆球從山上滾下來，在下坡的時候速度越來越快，遇到上坡，方向改變，速度下降」



- V_t ：「方向速度」，會跟上一次的更新有關

$$\textcircled{V_t} \leftarrow \beta V_{t-1} - \eta \frac{\partial L}{\partial w}$$

$$w \leftarrow w + V_t$$

- 如果上一次的梯度跟這次同方向的話， $|V_t|$ (速度)會越來越大(代表梯度增強)， w 參數的更新梯度便會越來越快，
- 如果方向不同， $|V_t|$ 便會比上次更小(梯度減弱)， w 參數的更新梯度便會變小



- 加入的這一項，可以使得梯度方向不變的維度上速度變快，梯度方向有所改變的維度上的更新速度變慢，這樣就可以加快收斂並減小震盪

Optimizer – SGD 說明

- SGD-隨機梯度下降法(stochastic gradient decent)
- 找出參數的梯度(利用微分的方法)，往梯度的方向去更新參數(weight) ，

SGD Weight update equation
$$w \leftarrow w - \eta \frac{\partial L}{\partial w}$$

w 為權重(weight)參數， L 為損失函數(loss function)， η 是學習率(learning rate)， $\partial L /$

∂w 是損失函數對參數的梯度(微分)

優點：SGD 每次更新時對每個樣本進行梯度更新，對於很大的數據集來說，可能會有相似的樣本，而 SGD 一次只進行一次更新，就沒有冗餘，而且比較快

缺點：但是 SGD 因為更新比較頻繁，會造成 cost function 有嚴重的震盪

Optimizer – SGD 調用

- `keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)`
 - `lr` : `<float>` 學習率。
 - Momentum 動量 : `<float>` 參數，用於加速 SGD 在相關方向上前進，並抑制震盪。
 - Decay(衰變) : `<float>` 每次參數更新後學習率衰減值。
 - `nesterov` : 布爾值。是否使用 Nesterov 動量。

```
from keras import optimizers
```

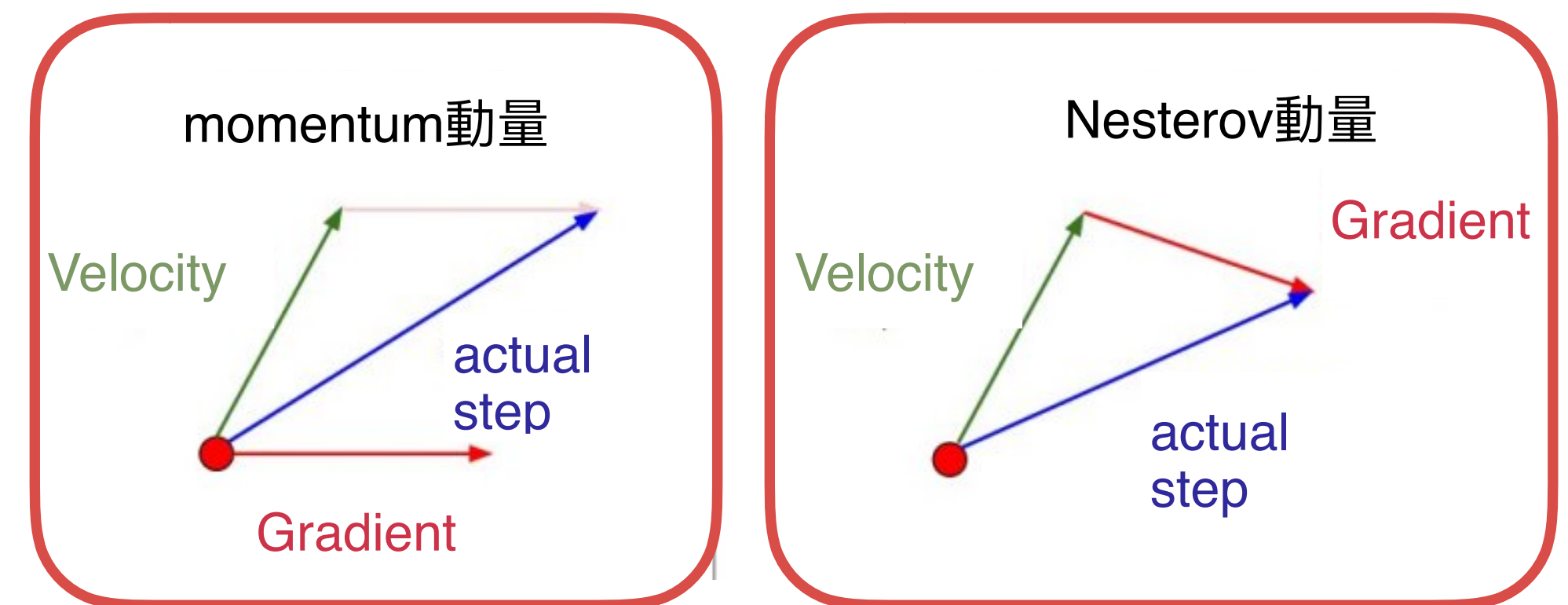
```
model = Sequential()  
model.add(Dense(64, kernel_initializer='uniform', input_shape=(10,)))  
model.add(Activation('softmax'))
```

#實例化一個優化器對象，然後將它傳入`model.compile()`，可以修改參數

```
sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)  
model.compile(loss='mean_squared_error', optimizer=sgd)
```

通過名稱來調用優化器，將使用優化器的默認參數。

```
model.compile(loss='mean_squared_error', optimizer='sgd')
```



圖片來源：cs231n.github

Optimizer – SGD, mini-batch gradient descent

- batch-gradient，其實就是普通的梯度下降算法但是採用批量處理。
 - 當數據集很大（比如有100000個左右時），每次 iteration 都要將1000000 個數據跑一遍，機器帶不動。於是有了 mini-batch-gradient ——將 1000000 個樣本分成 1000 份，每份 1000 個，都看成一組獨立的數據集，進行 forward_propagation 和 backward_propagation。
- 在整個算法的流程中，cost function 是局部的，但是W和b是全局的。
 - 批量梯度下降對訓練集上每一個數據都計算誤差，但只在所有訓練數據計算完成後才更新模型。
 - 對訓練集上的一次訓練過程稱為一代（epoch）。因此，批量梯度下降是在每一個訓練 epoch 之後更新模型。

Optimizer – SGD, mini-batch gradient descent

- epoch、iteration、batchsize, mini-batch
- batchsize：批量大小，即每次訓練在訓練集中取 batchsize 個樣本訓練；
 - batchsize=1;
 - batchsize = mini-batch;
 - batchsize = whole training set
- iteration：1 個 iteration 等於使用 batchsize 個樣本訓練一次；
- epoch：1 個 epoch 等於使用訓練集中的全部樣本訓練一次；

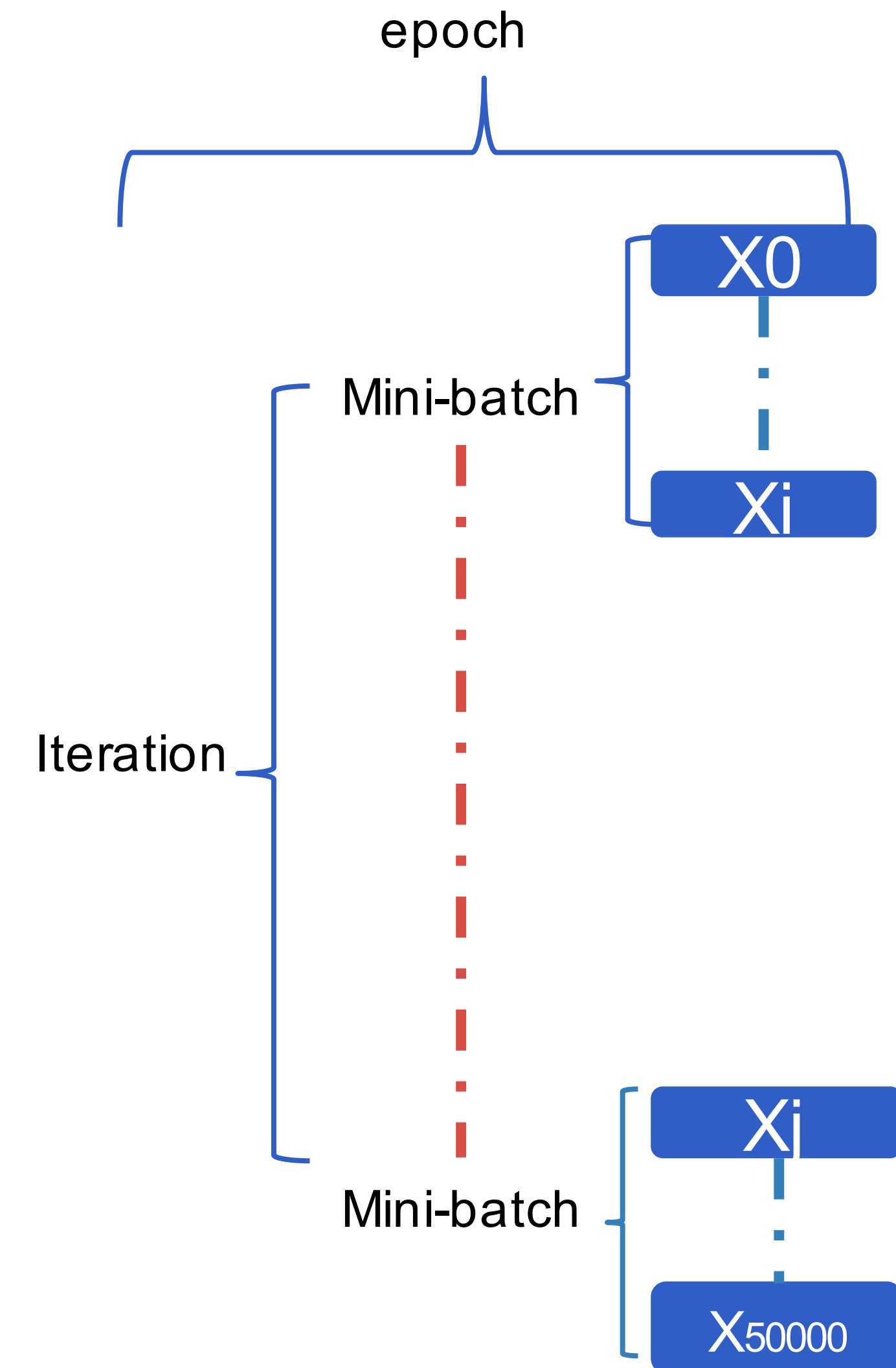
Example:

features is (50000, 400)

labels is (50000, 10)

batch_size is 128

Iteration = $50000/128+1 = 391$



Optimizer – SGD, mini-batch gradient descent



- 怎麼配置mini-batch梯度下降
 - Mini-batch sizes，簡稱為「batch sizes」，是算法設計中需要調節的參數。
 - 較小的值讓學習過程收斂更快，但是產生更多噪聲。
 - 較大的值讓學習過程收斂較慢，但是準確的估計誤差梯度。
 - batch size 的默認值最好是 32 盡量選擇 2 的冪次方，有利於 GPU 的加速。
 - 調節 batch size 時，最好觀察模型在不同 batch size 下的訓練時間和驗證誤差的學習曲線。
 - 調整其他所有超參數之後再調整 batch size 和學習率。

Optimizer - Adagrad

- 對於常見的數據給予比較小的學習率去調整參數，對於不常見的數據給予比較大的學習率調整參數
 - 每個參數都有不同的 learning rate,
 - 根據之前所有 gradient 的 root mean square 修改

第 t 次更新

$$g^t = \frac{\partial L}{\partial \theta} \Big|_{\theta=\theta^t}$$

Gradient

網路參數(b, w)

Gradient descent

$$\theta^{t+1} = \theta^t - \eta g^t$$

Adagrad

$$\theta^{t+1} = \theta^t - \frac{\eta}{\sigma^t} g^t$$

Learning rate

$$\sigma^t = \sqrt{\frac{(g^0)^2 + \dots + (g^t)^2}{t + 1}}$$

Root mean square(RMS)
of all Gradient

- 優點：Adagrad 的是減少了學習率的手動調節
- 缺點：它的缺點是分母會不斷積累，這樣學習率就會收縮並最終會變得非常小。

Optimizer – Adagrad 調用

- 超參數設定值:

`keras.optimizers.Adagrad(lr=0.01, epsilon=None, decay=0.0)`

- `lr` : float ≥ 0 . 學習率. 一般 η 就取 0.01
- `epsilon` : float ≥ 0 . 若為 None, 默認為 `K.epsilon()`.
- `decay` : float ≥ 0 . 每次參數更新後學習率衰減值

```
from keras import optimizers
```

```
model = Sequential()  
model.add(Dense(64, kernel_initializer='uniform', input_shape=(10,)))  
model.add(Activation('softmax'))
```

#實例化一個優化器對象，然後將它傳入`model.compile()`，可以修改參數

```
opt = optimizers.Adagrad(lr=0.01, epsilon=None, decay=0.0)  
model.compile(loss='mean_squared_error', optimizer=opt)
```

Optimizer - RMSprop

- RMSProp 算法也旨在抑制梯度的鋸齒下降，但與動量相比，RMSProp 不需要手動配置學習率超參數，由算法自動完成。更重要的是，RMSProp 可以為每個參數選擇不同的學習率。
- RMSprop 是為了解決 Adagrad 學習率急劇下降問題的，所以
- 比對梯度更新規則：

Adagrad

$$\theta^{t+1} = \theta^t - \frac{\eta}{\sigma^t} g^t$$
$$\sigma^t = \sqrt{\frac{(g^0)^2 + \dots + (g^t)^2}{t + 1}}$$

Root mean square (RMS) of all Gradient

RMSprop

$$\theta^{t+1} = \theta^t - \frac{\eta}{\sqrt{r^t}} g^t$$
$$r^t = (1-p)(g^t)^2 + pr^{t-1}$$

分母換成了過去的梯度平方的衰減平均值

Optimizer – RMSprop 調用

- `keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)`
- This optimizer is usually a good choice for recurrent neural networks.

Arguments

- `lr` : float ≥ 0 . Learning rate.
- `rho` : float ≥ 0 .
- `epsilon` : float ≥ 0 . Fuzz factor. If None, defaults to `K.epsilon()`.
- `decay` : float ≥ 0 . Learning rate decay over each update.

```
from keras import optimizers
```

```
model = Sequential()  
model.add(Dense(64, kernel_initializer='uniform', input_shape=(10,)))  
model.add(Activation('softmax'))
```

#實例化一個優化器對象，然後將它傳入`model.compile()`，可以修改參數

```
opt = optimizers.RMSprop(lr=0.001, epsilon=None, decay=0.0)  
model.compile(loss='mean_squared_error', optimizer=opt)
```

Optimizer – Adam 說明

- 除了像 RMSprop 一樣存儲了過去梯度的平方 v_t 的指數衰減平均值，也像 momentum 一樣保持了過去梯度 m_t 的指數衰減平均值，「 t 」：

the first moment (the mean) $m_t = \beta_1 m_t + (1 - \beta_1) g_t$

the second moment (the uncentered variance) $v_t = \beta_2 v_t + (1 - \beta_2) g_t^2$

- 計算梯度的指數移動平均數， m_0 初始化為 0。綜合考慮之前時間步的梯度動量。
- β_1 係數為指數衰減率，控制權重分配（動量與當前梯度），通常取接近於1的值。默認為 0.9
- 其次，計算梯度平方的指數移動平均數， v_0 初始化為 0。 β_2 係數為指數衰減率，控制之前的梯度平方的影響情況。類似於 RMSProp 算法，對梯度平方進行加權均值。默認為 0.999

Optimizer – Adam 說明

- 由於 m_0 初始化為 0，會導致 m_t 偏向於 0，尤其在訓練初期階段。所以，此處需要對梯度均值 m_t 進行偏差糾正，降低偏差對訓練初期的影響。
- 與 m_0 類似，因為 v_0 初始化為 0 導致訓練初始階段 v_t 偏向 0，對其進行糾正。

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

the decay rates are small
(i.e. β_1 and β_2 are close to 1)

更新參數，初始的學習率 lr 乘以梯度均值與梯度方差的平方根之比。其中默認學習率 $lr = 0.001$, $\epsilon = 10^{-8}$ ，避免除數變為 0。

對更新的步長計算，能夠從梯度均值及梯度平方兩個角度進行自適應地調節，而不是直接由當前梯度決定

Optimizer - Adam 調用

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0,  
amsgrad=False)
```

- lr : float ≥ 0 . 學習率。
- beta_1 : float, $0 < \text{beta} < 1$. 通常接近於 1。
- beta_2 : float, $0 < \text{beta} < 1$. 通常接近於 1。
- epsilon : float ≥ 0 . 模糊因數. 若為 None, 默認為 K.epsilon()。
- amsgrad : boolean. 是否應用此演算法的 AMSGrad 變種，來自論文「On the Convergence of Adam and Beyond」
- decay : float ≥ 0 . 每次參數更新後學習率衰減值。

```
from keras import optimizers
```

```
model = Sequential()  
model.add(Dense(64, kernel_initializer='uniform', input_shape=(10,)))  
model.add(Activation('softmax'))
```

#實例化一個優化器對象，然後將它傳入 `model.compile()`，可以修改參數

```
opt = optimizers.Adam(lr=0.001, epsilon=None, decay=0.0)  
model.compile(loss='mean_squared_error', optimizer=opt)
```

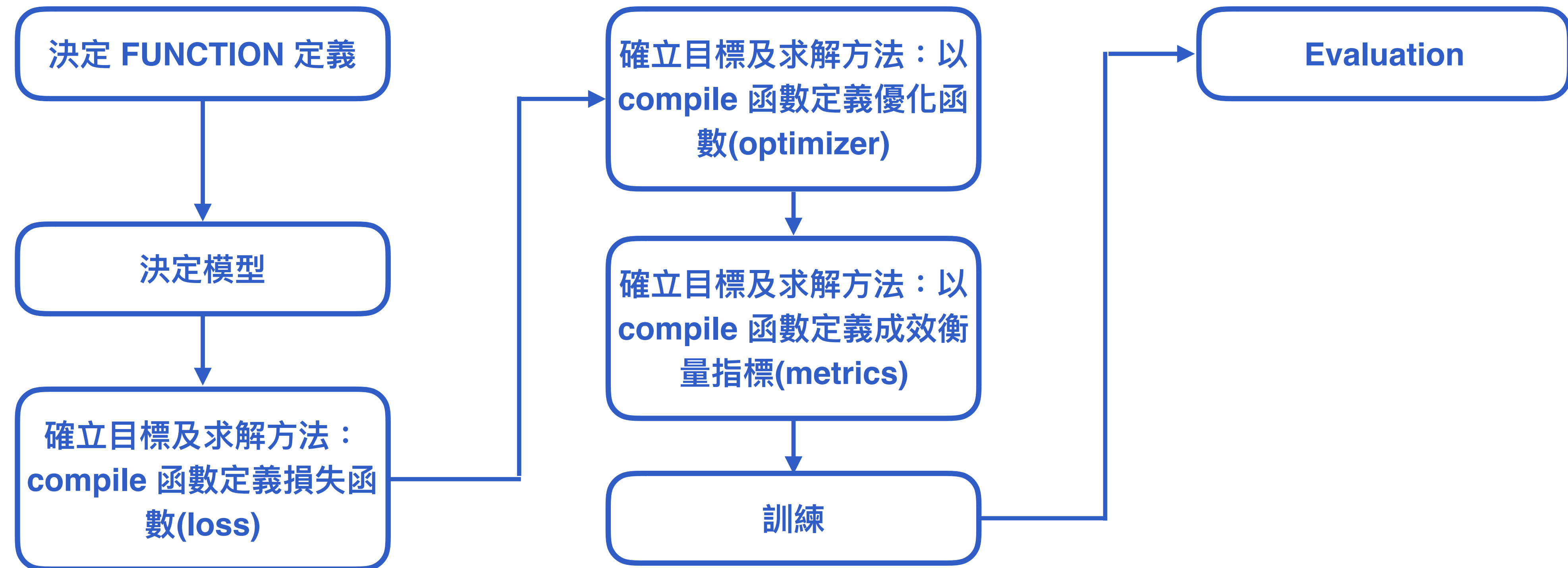
如何選擇優化器

- 隨機梯度下降 (SGD) : SGD 指的是 mini batch gradient descent 優點：針對大數據集，訓練速度很快。從訓練集樣本中隨機選取一個 batch 計算一次梯度，更新一次模型參數。
 - 缺點：
對所有參數使用相同的學習率。對於稀疏數據或特徵，希望盡快更新一些不經常出現的特徵，慢一些更新常出現的特徵。所以選擇合適的學習率比較困難。
- 容易收斂到局部最優 Adam : 利用梯度的一階矩估計和二階矩估計動態調節每個參數的學習率。
 - 優點：
 - 1.經過偏置校正後，每一次迭代都有確定的範圍，使得參數比較平穩。善於處理稀疏梯度和非平穩目標。
 - 2.對內存需求小
 - 3.對不同內存計算不同的學習率
- RMSProp : 自適應調節學習率。對學習率進行了約束，適合處理非平穩目標和 RNN。

如何選擇優化器

- 如果輸入數據集比較稀疏，SGD、NAG和動量項等方法可能效果不好。因此對於稀疏數據集，應該使用某種自適應學習率的方法，且另一好處為不需要人為調整學習率，使用默認參數就可能獲得最優值。
 - Adagrad, RMSprop, Adam。
- 如果想使訓練深層網絡模型快速收斂或所構建的神經網絡較為複雜，則應該使用Adam或其他自適應學習速率的方法，因為這些方法的實際效果更優。
 - Adam 就是在 RMSprop 的基礎上加了 bias-correction 和 momentum，
 - 隨著梯度變的稀疏，Adam 比 RMSprop 效果會好。

前述流程 / python程式 對照



前述流程 / python程式 對照

python 程式 (請參閱今日範例)

```
from keras import optimizers  
model = Sequential()
```

```
model.add(Dense(64, kernel_initializer='uniform', input_shape=(10,)))  
model.add(Activation('softmax'))
```

```
# set parameters:  
max_features = 5000  
batch_size = 32  
kernel_size = 3  
hidden_dims = 250  
epochs = 2
```

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])  
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,  
                    validation_data=(x_test, y_test))
```

重要知識點複習：動量Momentum

- 「一顆球從山上滾下來，在下坡的時候速度越來越快，遇到上坡，方向改變，速度下降」



- V_t ：「方向速度」，會跟上一次的更新有關

$$\textcircled{V_t} \leftarrow \beta V_{t-1} - \eta \frac{\partial L}{\partial w}$$

$$w \leftarrow w + V_t$$

- 如果上一次的梯度跟這次同方向的話， $|V_t|$ (速度)會越來越大(代表梯度增強)， w 參數的更新梯度便會越來越快，
- 如果方向不同， $|V_t|$ 便會比上次更小(梯度減弱)， w 參數的更新梯度便會變小



- 加入的這一項，可以使得梯度方向不變的維度上速度變快，梯度方向有所改變的維度上的更新速度變慢，這樣就可以加快收斂並減小震盪

重要知識點複習：mini-batch

- epoch、iteration、batchsize, mini-batch
- batchsize：批量大小，即每次訓練在訓練集中取 batchsize 個樣本訓練；
 - batchsize=1;
 - batchsize = mini-batch;
 - batchsize = whole training set
- iteration：1個 iteration 等於使用 batchsize 個樣本訓練一次；
- epoch：1個 epoch 等於使用訓練集中的全部樣本訓練一次；

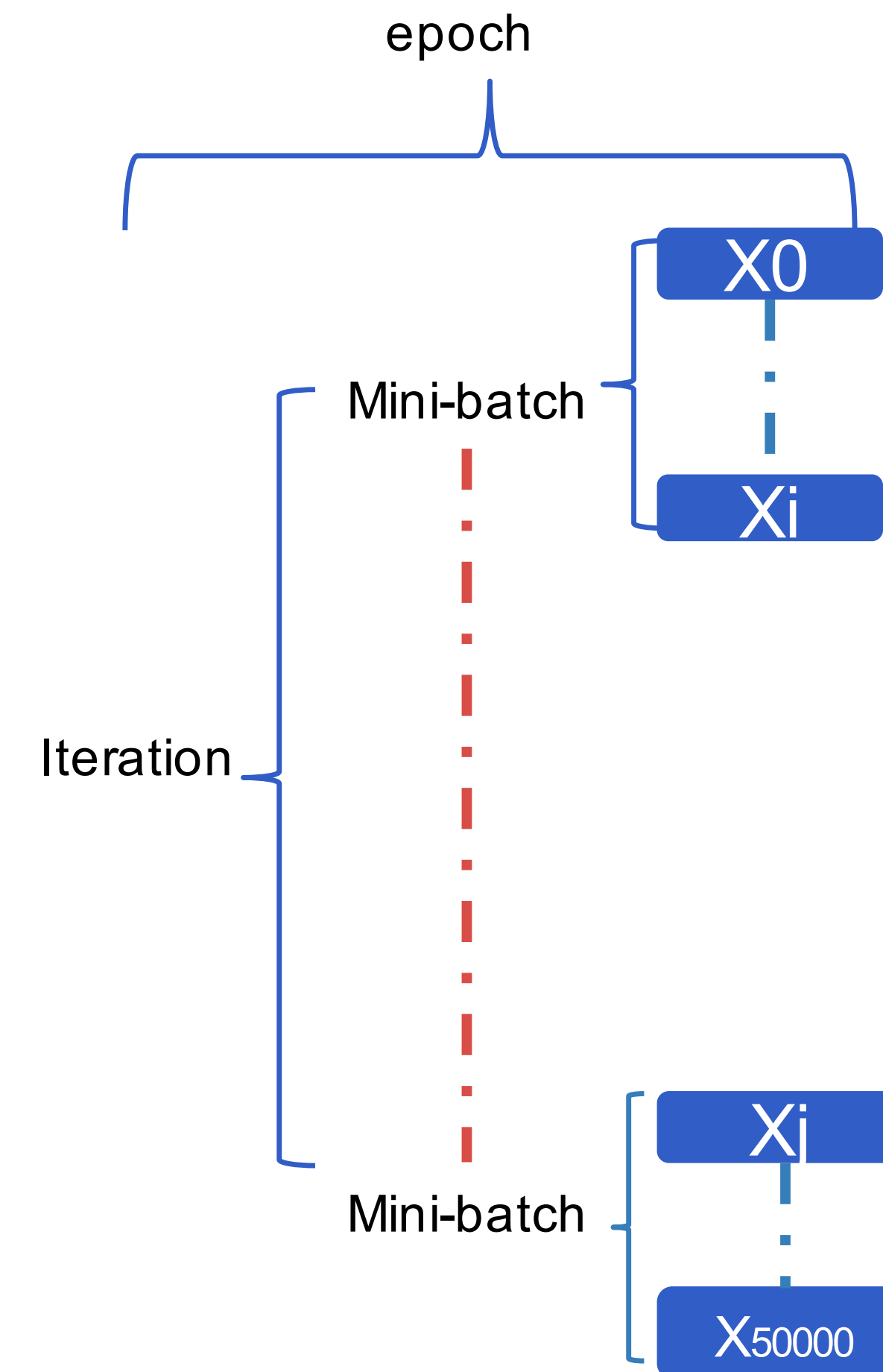
Example:

features is (50000, 400)

labels is (50000, 10)

batch_size is 128

Iteration = $50000/128+1 = 391$



重要知識點複習：幾個常用的優化器

- SGD (Stochastic Gradient Descent) 隨機梯度下降法這種方法是將數據分成一小批一小批的進行訓練。但是速度比較慢。
- AdaGrad 採用改變學習率的方式。
- RMSProp 這種方法是將 Momentum 與 AdaGrad 部分相結合。
- Adam，結合 AdaGrad 和 RMSProp 兩種優化算法的優點。對梯度的一階矩估計 (First Moment Estimation，即梯度的均值) 和二階矩估計 (Second Moment Estimation，即梯度的未中心化的方差) 進行綜合考慮，計算出更新步長。

解題時間 It's Your Turn

請跳出PDF至官網Sample Code & 作業
開始解題

