机器学习

实验四：神经网络模型

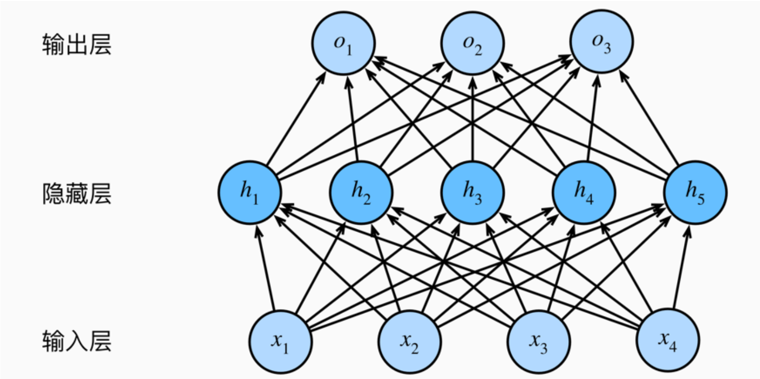
报告提交时间：6月18日23:59分前

提交方式：将压缩包发送至guo_cheng@bit.edu.cn

MLP

多层感知机

一种基础的前馈神经网络，由多个全连接层（Fully Connected Layers）组成，主要用于处理结构化数据和基础的分类、回归任务。



针对输入 $\mathbf{X} \in \mathbb{R}^{n\times d}$

隐藏层输出 $\mathbf{H} = \boxed{\sigma}\left(\mathbf{XW}^{(1)} + \mathbf{b}^{(1)}\right),$

输出层输出 $\mathbf{O} = \mathbf{HW}^{(2)} + \mathbf{b}^{(2)}.$

相较于线性变换，MLP中激活函数引入非线性便于网络学习稀疏化表示

*理论上可以逼近任意连续函数（Stone-Weierstrass 定理，任意连续函数都可以由简单函数线性组合逼近）

实验任务：

- 1. 使用分类模型（MLPClassifier）完成对手写数字的分类（load_digits)，并使用评测指标precision_score、recall_score、f1_score对分类结果评测
- 2.使用回归模型（ MLPRegressor ）实现对波士顿房价的预测(load_boston)，并使用r2-score 对回归结果评测
- 3.实现自定义神经网络回归模型，实现对自定义数据(lasso_data)的预测

实验环境：

- Python 3.0以上版本，开发工具任选。
- 使用scikit-learn包中的机器学习模型
- 安装说明：https://scikit-learn.org/stable/install.html#installation-instructions

实验要求和评分标准

- 完成实验中的3个项任务（前两个3分，后一个4分）

实验要求和评分标准

加分项：

- 使用GridSearchCV对实验任务一的分类模型调参，并将最佳参数和评分结果输出。（1分）
- 在实验任务一中采用不同激活函数（identity, logistic, tanh, relu）训练模型，并将评分结果输出。（1分）
- 任意选取load_boston 数据集中的2个特征，分别绘制部分依赖图（使用plot_partial_dependence）。（1分）

作业提交内容：

- 所有代码文件
- 每个模型运行后的结果截图（保存到word中，word文档的命名规则为：学号-姓名-实验4.docx）

实验步骤

https://scikit-learn.org/

- 加载数据集
- 拆分数据集
- 构建模型
- 获取在训练集中的模型
- 在测试集上预测结果
- 模型评测

数据集

datasets.load_boston([return_X_y])	Load and return the boston house-prices dataset (regression).
datasets.load_breast_cancer([return_X_y])	Load and return the breast cancer wisconsin dataset (classification).
datasets.load_diabetes([return_X_y])	Load and return the diabetes dataset (regression).
datasets.load_digits([n_class, return_X_y])	Load and return the digits dataset (classification).
datasets.load_files(container_path[, …])	Load text files with categories as subfolder names.
datasets.load_iris([return_X_y])	Load and return the iris dataset (classification).
datasets.load_linnerud([return_X_y])	Load and return the linnerud dataset (multivariate regression).
datasets.load_sample_image(image_name)	Load the numpy array of a single sample image
datasets.load_sample_images()	Load sample images for image manipulation.
datasets.load_svmlight_file(f[, n_features, …])	Load datasets in the svmlight / libsvm format into sparse CSR matrix
datasets.load_svmlight_files(files[, …])	Load dataset from multiple files in SVMlight format
datasets.load_wine([return_X_y])	Load and return the wine dataset (classification).

https://scikit-learn.org/stable/modules/classes.html#module-sklearn.datasets

分类模型

sklearn.neural_network.MLPClassifier

class `sklearn.neural_network.MLPClassifier`(*hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000*)

[source]

hidden_layer_sizes : *tuple, length = n_layers - 2, default=(100,)*

The ith element represents the number of neurons in the ith hidden layer.

activation : *{'identity', 'logistic', 'tanh', 'relu'}, default='relu'*

Activation function for the hidden layer.

- 'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x
- 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
- 'tanh', the hyperbolic tan function, returns f(x) = tanh(x).
- 'relu', the rectified linear unit function, returns f(x) = max(0, x)

Sigmoid

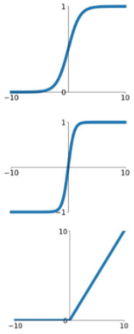
$\sigma(x) = \frac{1}{1+e^{-x}}$

tanh

$\tanh(x)$

ReLU

$\max(0, x)$



分类模型

sklearn.neural_network.MLPClassifier

class `sklearn.neural_network.MLPClassifier`*(hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsi*

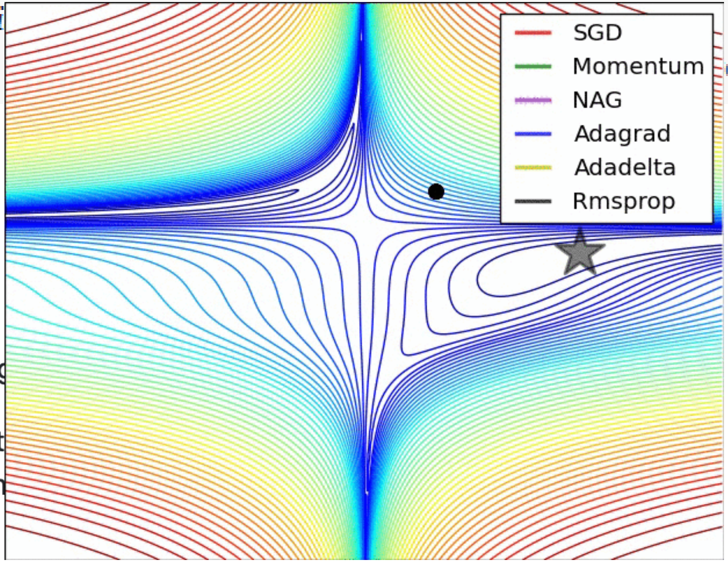
ce]

solver : *{'lbfgs', 'sgd', 'adam'}, default='adam'*

The solver for weight optimization.

- 'lbfgs' is an optimizer in the family of quasi-Newton methods.
- 'sgd' refers to stochastic gradient descent.
- 'adam' refers to a stochastic gradient-based optimizer proposed by King

Note: The default solver 'adam' works pretty well on relatively large dataset samples or more) in terms of both training time and validation score. For sm can converge faster and perform better.



使用MLPClassifier实现分类任务

In [119]:
```python
from sklearn.datasets import load_wine
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
```

In [123]:
```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

In [124]:
```python
models = (MLPClassifier(solver='lbfgs'),
          MLPClassifier(solver='lbfgs', hidden_layer_sizes=[10]),
          MLPClassifier(solver='lbfgs', hidden_layer_sizes=[10, 10]),
          MLPClassifier(solver='lbfgs', hidden_layer_sizes=[10, 10], activation='tanh')
)
models = (clf.fit(X_train, y_train) for clf in models)
```

In [125]:

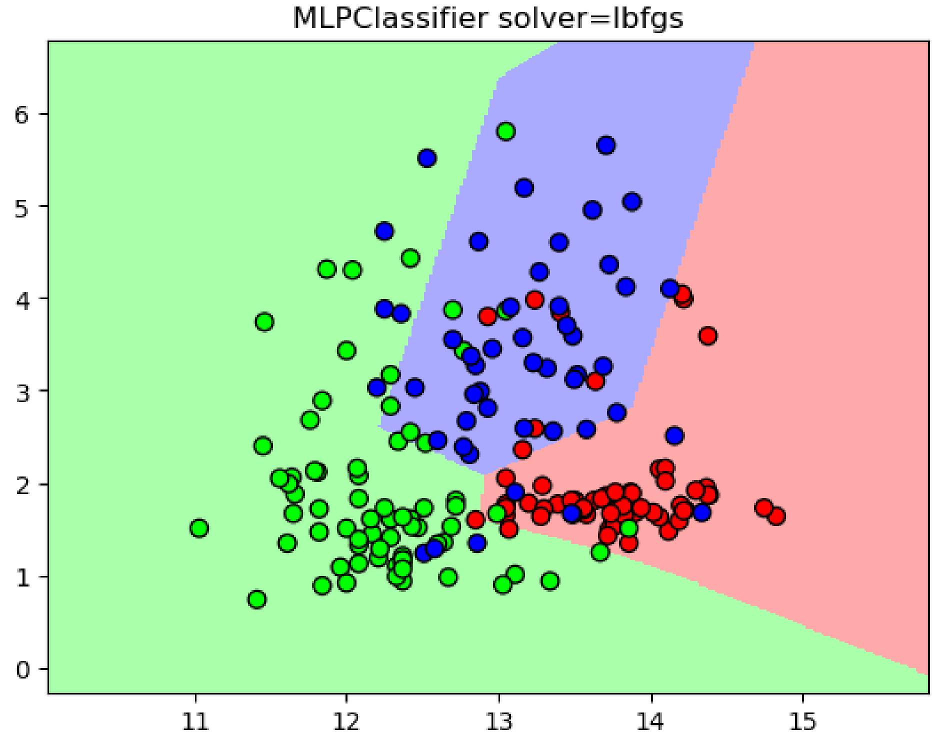
```python
titles = ('MLPClassifier, solver=lbfgs',
          'MLPClassifier, solver=lbfgs, hidden_layer_sizes=[10]',
          'MLPClassifier, solver=lbfgs, hidden_layer_sizes=[10, 10]',
          'MLPClassifier, solver=lbfgs, hidden_layer_sizes=[10, 10], activation=tanh'
)
```

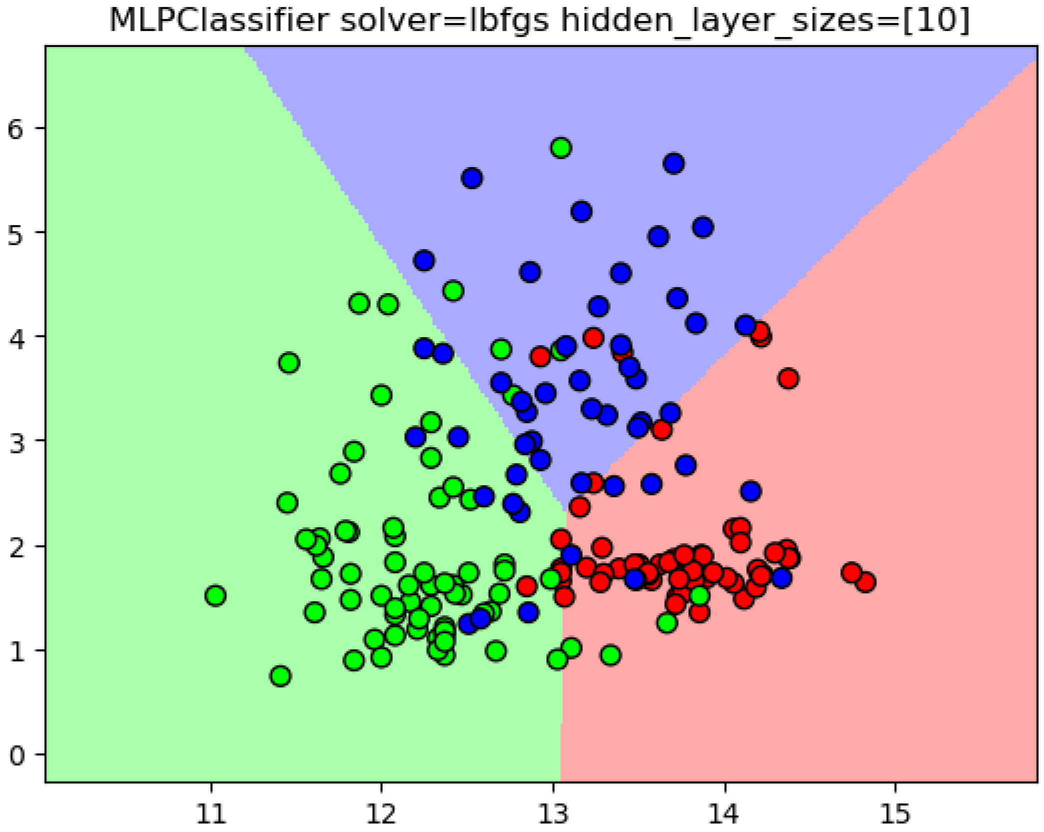
In [126]:

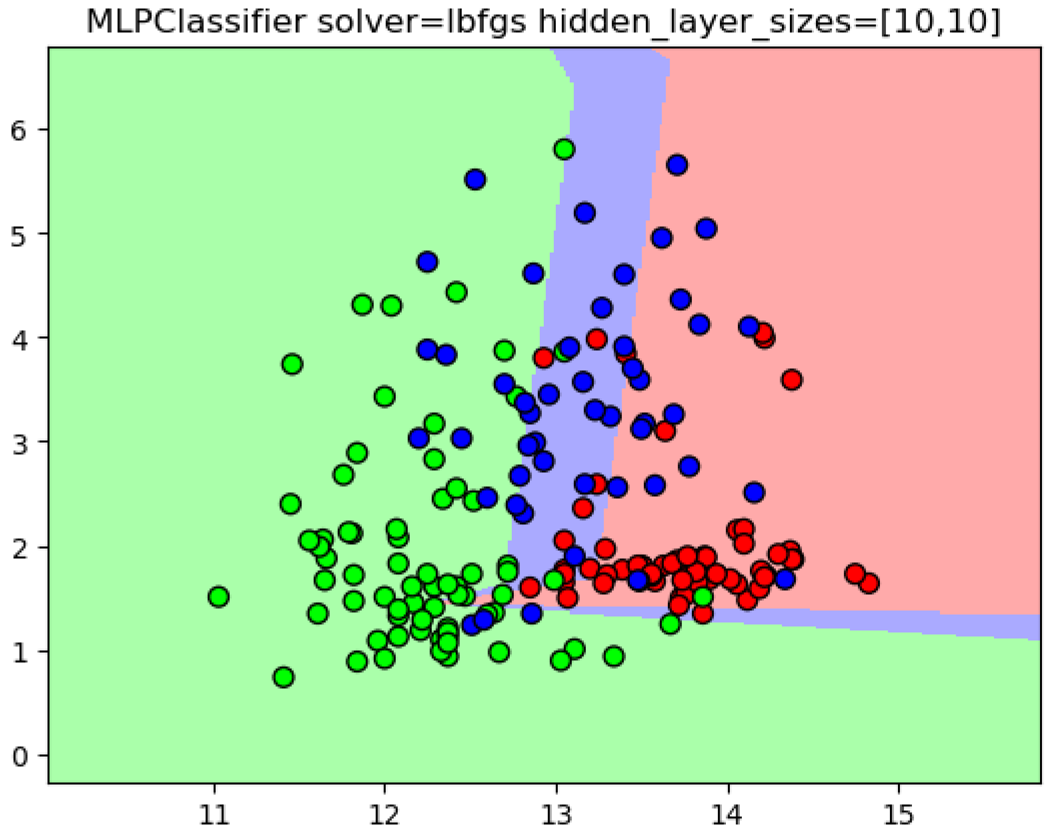
```python
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, .02),
                     np.arange(y_min, y_max, .02))
```

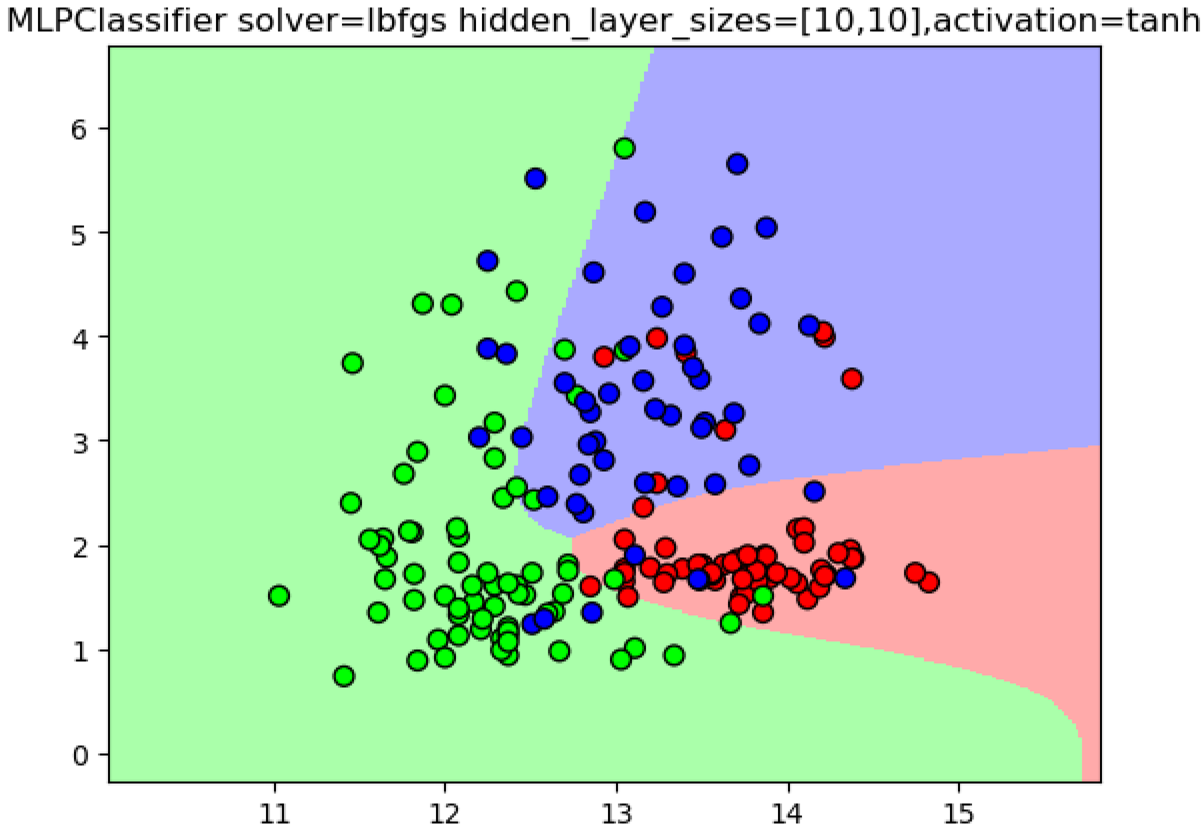
In [127]:

```python
fig, sub = plt.subplots(2, 2, figsize = (10, 6))
plt.subplots_adjust(wspace=2, hspace=0.6) #wspace, hspace：子图之间的横向间距、纵向间距分别与子图平均宽度、平均高度的比值。
for clf, title, ax in zip(models, titles, sub.flatten()):
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    ax.pcolormesh(xx, yy, Z, cmap=cmap_light)
    # 将数据特征用散点图表示出来
    ax.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', s=60)
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_title(title)
    ax.set_title(title)
plt.show()
```


MLPClassifier solver=lbfgs
6
5
4
3
2
1
0
11
12
13
14
15


MLPClassifier solver=lbfgs hidden_layer_sizes=[10]
6
5
4
3
2
1
0
11
12
13
14
15


MLPClassifier solver=lbfgs hidden_layer_sizes=[10,10]
0
1
2
3
4
5
6
11
12
13
14
15


MLPClassifier solver=lbfgs hidden_layer_sizes=[10,10],activation=tanh
0
1
2
3
4
5
6
11
12
13
14
15

回归模型

sklearn.neural_network.MLPRegressor

class `sklearn.neural_network.MLPRegressor`(*hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000*)

[source]

sklearn.inspection.plot_partial_dependence

sklearn.inspection.plot_partial_dependence(*estimator, X, features, *, feature_names=None, target=None, response_method='auto', n_cols=3, grid_resolution=100, percentiles=(0.05, 0.95), method='auto', n_jobs=None, verbose=0, line_kw=None, ice_lines_kw=None, pd_line_kw=None, contour_kw=None, ax=None, kind='average', subsample=1000, random_state=None*)

[source]

回归模型

In [40]:
```
from sklearn.inspection import plot_partial_dependence
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_boston
```

In [41]:
```
boston= load_boston()
boston.feature_names
```

Out[41]:
```
array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT'], dtype='<U7')
```

In [42]:
```
x=boston.data
y=boston.target
```

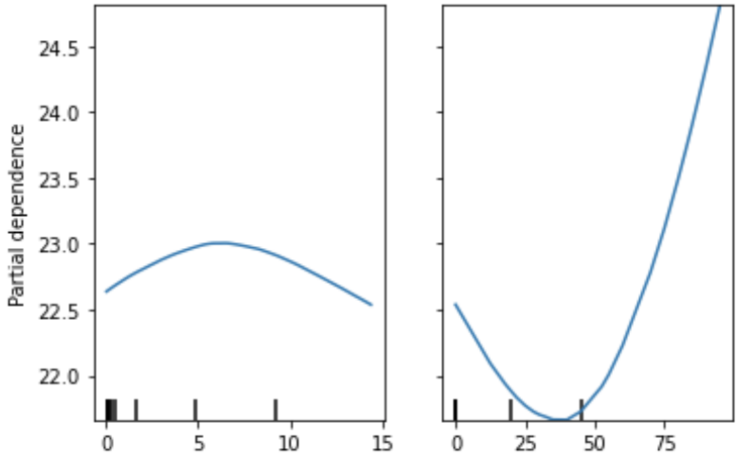
In [43]:
```
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
```

In [44]:
```
clf = MLPRegressor(hidden_layer_sizes=(50, 50), solver='lbfgs', max_iter=5000)
clf.fit(X_train, y_train)
```

Out[44]:
```
MLPRegressor(hidden_layer_sizes=(50, 50), max_iter=5000, solver='lbfgs')
```

In [55]:
```
features = [0, 1]
```

In [58]:
```
plot_partial_dependence(clf, X_train, features, feature_names=feature_names)
```



使用Keras实现神经网络模型

In [26]:
```
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizer_v1 import SGD
from keras.datasets import mnist
import numpy
from keras.utils import np_utils
```

In [27]:
```
# 1、加载数据
(x_train, y_train), (x_test, y_test)=mnist.load_data()
x_train.shape
```

Out[27]:
```
(60000, 28, 28)
```

In [28]:
```
x_test.shape
```

Out[28]:
```
(10000, 28, 28)
```

使用Keras实现神经网络模型

In [29]:
```
# 数据处理
x_train = x_train.reshape(60000, 784).astype('float64')
x_test = x_test.reshape(10000, 784).astype('float64')
x_train_normalize = x_train/255
x_test_normalize = x_test/255
y_train_onehot = np_utils.to_categorical(y_train)
y_test_onehot = np_utils.to_categorical(y_test)
```

In [30]:
```
# 2、构建网络
model = Sequential()
model.add(Dense(units=256, input_dim=784, activation='relu'))
model.add(Dense(units=10, activation='softmax'))
print(model.summary())
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 256)               200960

dense_9 (Dense)              (None, 10)                2570

=================================================================
Total params: 203,530
Trainable params: 203,530
Non-trainable params: 0
_________________________________________________________________
None
```

使用Keras实现神经网络模型

In [31]:
```
# 3、编译并训练模型
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(x_train_normalize, y_train_onehot, validation_split=0.2, batch_size=200,epochs=10, verbose=2)
```

```
Epoch 1/10
```

```
2022-04-10 12:58:12.923400: W tensorflow/core/framework/cpu_allocator_impl.cc:82] Allocation of 150528000 exceeds 10% of free system memor
y.
```

```
240/240 - 1s - loss: 0.3871 - accuracy: 0.8942 - val_loss: 0.2020 - val_accuracy: 0.9438 - 681ms/epoch - 3ms/step
Epoch 2/10
240/240 - 0s - loss: 0.1711 - accuracy: 0.9514 - val_loss: 0.1463 - val_accuracy: 0.9578 - 343ms/epoch - 1ms/step
Epoch 3/10
240/240 - 0s - loss: 0.1205 - accuracy: 0.9658 - val_loss: 0.1203 - val_accuracy: 0.9661 - 345ms/epoch - 1ms/step
Epoch 4/10
240/240 - 0s - loss: 0.0920 - accuracy: 0.9740 - val_loss: 0.1058 - val_accuracy: 0.9693 - 350ms/epoch - 1ms/step
Epoch 5/10
240/240 - 0s - loss: 0.0730 - accuracy: 0.9792 - val_loss: 0.1016 - val_accuracy: 0.9703 - 424ms/epoch - 2ms/step
Epoch 6/10
240/240 - 0s - loss: 0.0585 - accuracy: 0.9833 - val_loss: 0.0899 - val_accuracy: 0.9744 - 465ms/epoch - 2ms/step
Epoch 7/10
240/240 - 0s - loss: 0.0476 - accuracy: 0.9875 - val_loss: 0.0853 - val_accuracy: 0.9759 - 499ms/epoch - 2ms/step
Epoch 8/10
240/240 - 0s - loss: 0.0395 - accuracy: 0.9894 - val_loss: 0.0869 - val_accuracy: 0.9751 - 499ms/epoch - 2ms/step
Epoch 9/10
240/240 - 1s - loss: 0.0321 - accuracy: 0.9923 - val_loss: 0.0828 - val_accuracy: 0.9755 - 565ms/epoch - 2ms/step
Epoch 10/10
240/240 - 1s - loss: 0.0268 - accuracy: 0.9937 - val_loss: 0.0833 - val_accuracy: 0.9758 - 536ms/epoch - 2ms/step
```

Out[31]:
```
<keras.callbacks.History at 0x7f182ecc3f40>
```

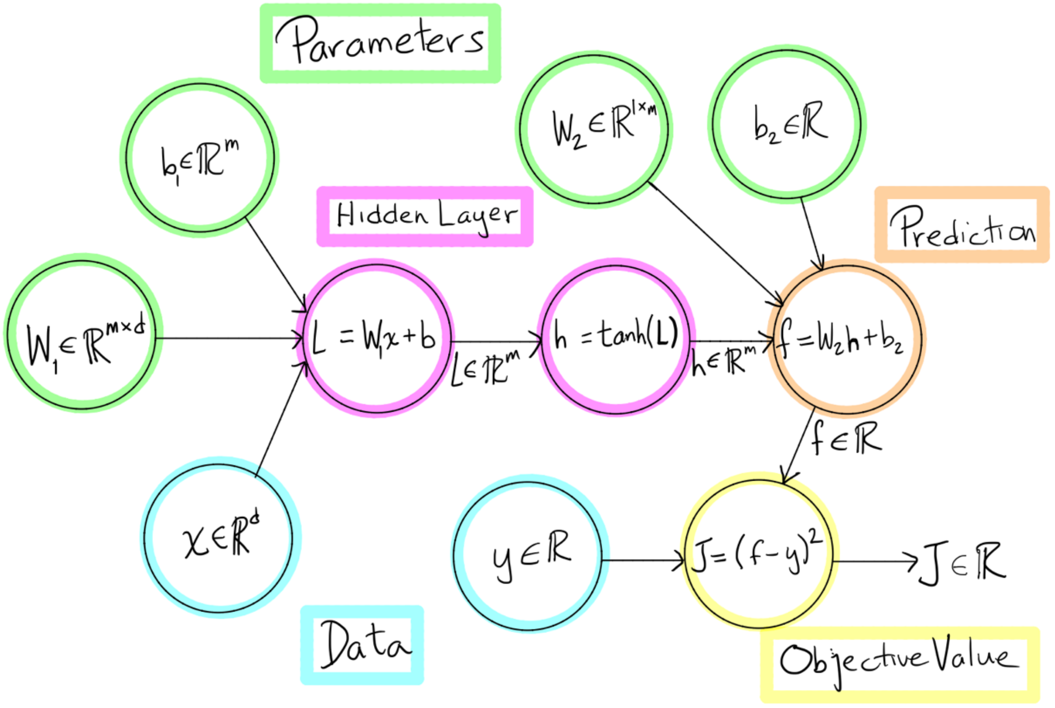
使用Keras实现神经网络模型

In [33]:
```
# 4、评估模型
score = model.evaluate(x_test_normalize, y_test_onehot)
print('accuracy = ',score[1])
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0652 - accuracy: 0.9811
accuracy =  0.9811000227928162
```

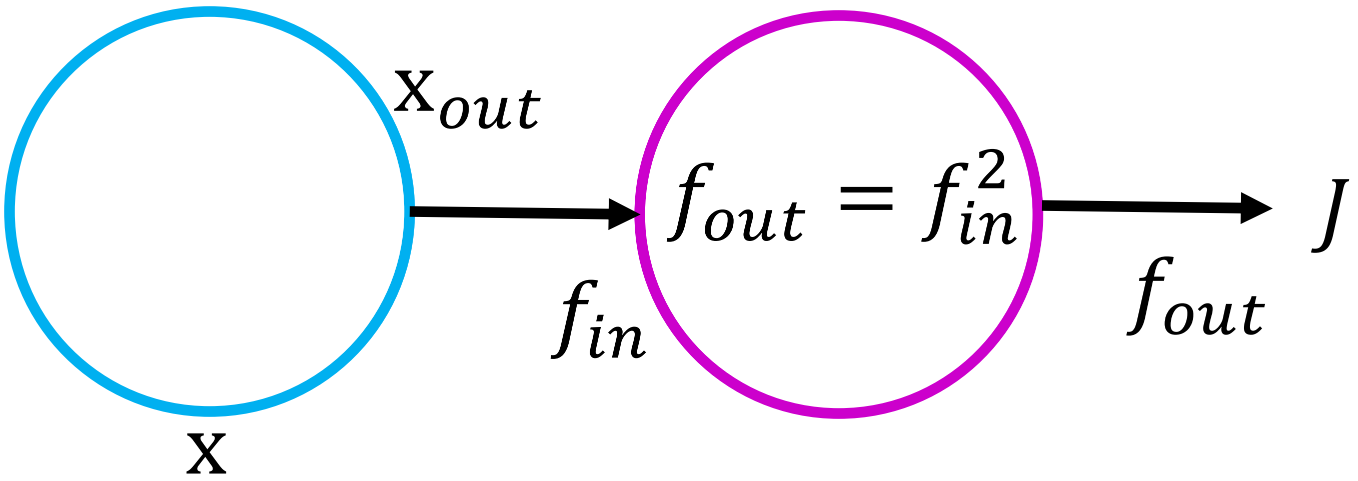
自定义实现神经网络

■计算图实现-data/hidden layer/prediction



自定义实现神经网络

- $f=x^2$例子



自定义实现神经网络

- $f=x^2$例子

```
class SquareNode(object):
    def __init__(self, x, node_name):
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.x = x

    def forward(self):
        self.out = self.x.out ** 2
        self.d_out = np.zeros(self.out.shape)
        return self.out
```

```
    def backward(self):
        d_x = self.d_out * 2 * self.x.out
        self.x.d_out += d_x
        return self.d_out

    def get_predecessors(self):
        return [self.x]
```

自定义实现神经网络

f=x^2例子

$$\frac{\partial J}{\partial f_{\text{in}}} = \left(\frac{\partial J}{\partial f_{\text{out}}}\right)\left(\frac{\partial f_{\text{out}}}{\partial f_{\text{in}}}\right)$$
$$= \left(\texttt{self.d\_out}\right)\left(2 \cdot \texttt{self.x.out}\right)$$

```python
# 构建计算图
x = ValueNode("x") # 输入节点
f = SquareNode(x, "Squaring Node")  #输出节点
# 设置初始值
x.out = np.array(7)
# 计算前向传播
x.forward()
f.forward() # 返回计算图输出
```

```
Out[7]:   49
```

自定义实现神经网络

- $f=x^2$例子

$$\frac{\partial J}{\partial f_{\text{in}}} = \left(\frac{\partial J}{\partial f_{\text{out}}}\right)\left(\frac{\partial f_{\text{out}}}{\partial f_{\text{in}}}\right) = \left(\texttt{self.d\_out}\right)\left(2\cdot\texttt{self.x.out}\right)$$

```
f.d_out = np.array(1) #初始化反向传播
f.backward()
x.backward()
```

```
Out[8]:   array(14.0)
```

自定义实现神经网络

f=x^2例子

$$\frac{\partial J}{\partial f_{\text{in}}} = \left(\frac{\partial J}{\partial f_{\text{out}}}\right)\left(\frac{\partial f_{\text{out}}}{\partial f_{\text{in}}}\right)$$

$$(\texttt{self.d\_out})\,(2\cdot\texttt{self.x.out})$$

```python
obj_vals = []
x_vals = []
num_steps = 15
step_size = .3
x.out = np.array(3)  # 设置初始值为3
for step_num in range(num_steps):
    x_vals.append(x.out) # 保存每一步的输入
    # 前向
    x.forward()
    J = f.forward() # 目标函数
    print("x=",x.out,"J(x)=",J)
    obj_vals.append(J) # 保存目标函数值
    # 后向
    f.d_out = np.array(1) #初始后向传播
    f.backward()
    x_grad = x.backward() # 保存梯度
    # 梯度下降
    x.out = x.out - step_size * x_grad
```

```
x= 3 J(x)= 9
x= 1.2 J(x)= 1.44
x= 0.48 J(x)= 0.2304
x= 0.192 J(x)= 0.036864
x= 0.0768 J(x)= 0.00589824
x= 0.03072 J(x)= 0.0009437184
x= 0.012288 J(x)= 0.000150994944
x= 0.0049152 J(x)= 2.415919104e-05
x= 0.00196608 J(x)= 3.8654705664e-06
x= 0.000786432 J(x)= 6.18475290624e-07
x= 0.0003145728 J(x)= 9.89560464998e-08
x= 0.00012582912 J(x)= 1.583296744e-08
x= 5.0331648e-05 J(x)= 2.5332747904e-09
x= 2.01326592e-05 J(x)= 4.05323966463e-10
x= 8.05306368e-06 J(x)= 6.48518346341e-11
```

自定义神经网络

文件说明

- test_utils.py
- setup_problem.py
- plot_utils.py
- nodes.py
- mlp_regression.t.py
- mlp_regression.py
- lasso_data.pickle
- graph.py

对应实现node.py与mlp_regression.py中TODO内容

自定义神经网络

◻样例输出