模型介绍

吴清柳

Beijing University of Posts and Telecommunications

August 18, 2023

Table of Contentes

使用机器学习方法预测从泰坦尼克灾难中生存的可能

Overview

通过该 ppt，对 9 个实验使用的模型进行介绍．九个实验分别是鸢尾花分类实验，emojify 人脸表情识别实验，贷款预测，房价预测，MNIST 手写数字识别，股票价格预测，泰坦尼克号生还预测，红酒质量预测和假新闻预测．
使用的模型有机器学习方法如决策树，逻辑回归，XGBoost 等；以及深度学习方法如全连接网络，CNN 等．

使用机器学习方法预测从泰坦尼克灾难中生存的可能

鸢尾花分类实验

代码和数据集

该论文的复现代码
在https://github.com/Micuks/code/blob/master/gnn/1_iris/iris-classification/train.ipynb 数据集使用了鸢尾花分类数据集.

模型信息

模型使用 PyTorch 编写, 为三层的全连接网络. 前两层使用 ReLU 激活函数, 第三层为完成分类功能, 使用 Softmax 作为激活函数, 输出分类结果.

- 第一层全连接层输入特征数量为 4, 输出特征数量为 25;
- 第二层全连接层输入特征数量为 25, 输出特征数量为 30;
- 第三层全连接层输入特征数量为 30, 输出特征数量为 3, 对应三类鸢尾花;

模型信息

模型搭建代码如下.

```
class Model(nn.Module):
def __init__(self, input_feats=4, hidden_layer1=25,
 ↪    hidden_layer2=30, output_feats=3) -> None:
    super().__init__()
    self.fc1 = nn.Linear(input_feats, hidden_layer1)
    self.fc2=nn.Linear(hidden_layer1, hidden_layer2)
    self.out=nn.Linear(hidden_layer2, output_feats)

def forward(self, x):
    x=F.relu(self.fc1(x))
    x=F.relu(self.fc2(x))
    x=self.out(x)

    return x
```

模型介绍

输入数据为鸢尾花 (Iris) 的特征数据. 鸢尾花有三种, 分别是 Setosa, Versicolour, Virginica. 每个都有 4 个特征: sepal length, sepal width, 以及 petal width.
模型的结构介绍如下.

模型结构

1. **输入层**神经元数量为 4, 代表 4 个输入特征.
2. **第一个隐藏层 'fc1'** 全连接层 (dense layer), 25 个神经元, 激活函数为 ReLU, 当输入为正时输出输入本身, 否则输出 0. 如果没有非线性激活函数, 则模型将等效为线性拟合函数, 拟合性能下降.
3. **第二个隐藏层 'fc2'** 另一个全连接层, 30 个神经元, 激活函数为 ReLU;
4. **输出层 'out'** 一个全连接层, 3 个神经元, 对应鸢尾花的三个分类: Setosa, Versicolour, Virginica.

训练方法

使用交叉熵作为损失函数，Adam 为优化器，学习率为 0.01，在训练集上进行 100 轮训练．对 loss 可视化观察后看到大约 40 轮后模型已经接近收敛．

训练代码

```
epochs=100
losses=[]
for i in range(epochs):
    y_pred=model.forward(X_train)
    loss=criterion(y_pred,y_train)
    losses.append(loss)
    print(f'epoch: {i:2} loss: {loss.item():10.8f}')

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

初始化

- 一轮代表对所有训练样本的一个完全的前向传播和反向传播过程;
- *losses* 列表会存储在每轮中计算的 loss 值，用于追踪模型的表现;

$$epochs = 100$$
$$losses = []$$

训练循环：前向传播和损失计算

Within the loop, the model predicts outcomes and computes the discrepancy between predictions and actual values. 在训练循环中，模型预测输出，计算预测值和实际值之间的差异;

```
for i in range(epochs):
    y_pred = model.forward(X_train)
    loss = criterion(y_pred, y_train)
    losses.append(loss)
```

训练循环：日志和优化

- 通过打印轮数和 loss 值来监控训练进程;
- 计算梯度并更新模型参数来降低 loss;

```
print(f'epoch: {i:2} loss: {loss.item():10.8f}')
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

使用机器学习方法预测从泰坦尼克灾难中生存的可能

利用 CNN 进行面部表情识别

在该实验中，利用 Tensorflow 来搭建 CNN 网络进行对摄像头采集的人脸进行表情识别. 模型使用 GPU 进行训练，得到的权重文件作为表情识别的依据. 此处使用 Tensorflow 是为了将 Pytorch 和 Tensorflow 两个常用框架都进行学习.

CNN 模型结构

- 目的：从灰度图片进行表情识别;
- 图片输入形状：$48 \times 48 \times 1$;
- 层之间顺序堆叠;

$$\textit{emotion\_model} = \textit{Sequential()}$$

Keras 中的 'Sequential' 是层之间的线性堆叠. 可以允许我们来通过将层逐个堆叠来搭建神经网络.

初始化卷积层

```
emotion_model.add(
    Conv2D(32, kernel_size=(3, 3), activation="relu",
    ↪    input_shape=(48, 48, 1))
)
emotion_model.add(Conv2D(64, kernel_size=(3, 3),
 ↪    activation="relu"))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Dropout(0.25))
```

第一个卷积层

第一个卷积层有 32 个卷积核，每个尺寸为 3×3，来处理尺寸为 48×48 的输入图片．‘relu’激活函数引入了非线性，让模型可以捕获复杂的特征．

初始化卷积层

```
emotion_model.add(
    Conv2D(32, kernel_size=(3, 3), activation="relu",
    ↪    input_shape=(48, 48, 1))
)
emotion_model.add(Conv2D(64, kernel_size=(3, 3),
 ↪    activation="relu"))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Dropout(0.25))
```

第二个卷积层

第二个卷积层有 64 个 3×3 的卷积核，该层进一步处理了来自前一层的特征图，检测了输入图片中的更复杂的特征.

初始化卷积层

```
emotion_model.add(
    Conv2D(32, kernel_size=(3, 3), activation="relu",
    ↪    input_shape=(48, 48, 1))
)
emotion_model.add(Conv2D(64, kernel_size=(3, 3),
 ↪    activation="relu"))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Dropout(0.25))
```

Max-Pooling 层

该层通过从每个 2×2 窗口中取最大值来对特征图进行降采样. 这减少了计算负担, 并帮助让模型平移不变.

初始化卷积层

```
emotion_model.add(
    Conv2D(32, kernel_size=(3, 3), activation="relu",
    ↪    input_shape=(48, 48, 1))
)
emotion_model.add(Conv2D(64, kernel_size=(3, 3),
 ↪    activation="relu"))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Dropout(0.25))
```

Dropout 层

Dropout 可以用来避免过拟合. 0.25 的 Dropout 率意味着大约 25% 的神经元在前面的层会被随机在训练中关闭, 使得可以获得容错率更高的模型.

多个卷积层

```
emotion_model.add(Conv2D(128, kernel_size=(3, 3),
    activation="relu"))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Conv2D(128, kernel_size=(3, 3),
    activation="relu"))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Dropout(0.25))
```

更深的卷积层

后面的具有 128 个卷积核的卷积层每个都进一步细化了特征图. 随着模型变深, 这些层会捕获图片中更多的高层特征.

每个卷积层后面都有一个 Max-Pooling 层来降采样, 降低计算负担, 提高平移不变性.

全连接层

```
emotion_model.add(Flatten())
emotion_model.add(Dense(1024, activation="relu"))
emotion_model.add(Dropout(0.25))
emotion_model.add(Dense(7, activation="softmax"))
```

Flatten 扁平化层

该层将 2D 的特征矩阵映射到 1D 的向量. 在将数据传入全连接层之前, 这是必要的步骤;

全连接层

```
emotion_model.add(Flatten())
emotion_model.add(Dense(1024, activation="relu"))
emotion_model.add(Dropout(0.25))
emotion_model.add(Dense(7, activation="softmax"))
```

稠密全连接层

具有 1024 个神经元的全连接层允许模型来基于由卷积层提取的高层次特征来进行决策. ‘ReLU’ 激活函数确保了非线性.

全连接层

```
emotion_model.add(Flatten())
emotion_model.add(Dense(1024, activation="relu"))
emotion_model.add(Dropout(0.25))
emotion_model.add(Dense(7, activation="softmax"))
```

输出层

输出稠密层有 7 个神经元，对应在数据集中的七种表情分类．‘softmax’激活函数确保了输出值是和为 1 的概率．

全连接层

```
emotion_model.add(Flatten())
emotion_model.add(Dense(1024, activation="relu"))
emotion_model.add(Dropout(0.25))
emotion_model.add(Dense(7, activation="softmax"))
```

Softmax

Softmax 函数常常被用在基于神经网络的分类器的最后一层，将一个实数向量转化为一个概率分布.

在数学上，对一个向量 z 的 Softmax 函数 σ 可以被定义为

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{1}$$

其中，

- $\sigma(z)_i$ 是 z 的第 i^{th} 个分量的 Softmax 函数输出;
- K 是分类的数量 (也是向量 z 的长度);

全连接层

```
emotion_model.add(Flatten())
emotion_model.add(Dense(1024, activation="relu"))
emotion_model.add(Dropout(0.25))
emotion_model.add(Dense(7, activation="softmax"))
```

Softmax

指数函数确保了输出向量所有的分量都是非负的，除法将所有的分量规范化，让他们的和为 1．结果向量因此可以表示 K 个类别的概率分布．

全连接层

```
emotion_model.add(Flatten())
emotion_model.add(Dense(1024, activation="relu"))
emotion_model.add(Dropout(0.25))
emotion_model.add(Dense(7, activation="softmax"))
```

为什么使用 Softmax

1. **概率解释** softmax 函数被用于分类模型的输出层，因为他提供了多个类之间的概率分布．这意味着对于给定的输入，模型可以提供输入可能属于哪个输出的类别;
2. **处理多个类** 在分类问题中，尤其是对于超过两个类别的情况，分类时需要不仅知道哪个类是最可能属于的，还想预测和其他类的似然成都．Softmax 可以通过将每个类的输出压缩在 0 和 1 之间，并确保他们的和为 1 来帮助实现这一目标．
3. **基于梯度的优化** Softmax 与类别交叉熵作为 loss 函数，提供了一个表现良好的梯度．在训练神经网络的反向传播阶段非常有用．没有良好的梯度，神经网络将不能高效的学习．

Cross-Entropy

Cross-entropy 是衡量两个概率分布之间的差异的方法. 在机器学习和深度学习中, 它常常被用作损失函数来量化预测概率分布和真实分布之间的差异.
对两个离散的概率分布 p 和 q, 交叉熵 $H(p,q)$ 被定义为

$$H(p,q) = -\sum_{x} p(x)\log\left(q(x)\right) \tag{1}$$

其中,

- $p(x)$ 是事件 x 发生的真实概率;
- $q(x)$ 是事件 x 发生的预测概率;
- 对于所有的可能事件 x 进行求和;

对于用于分类任务的神经网络, p 是真实标签的独热编码向量, q 是预测概率 (经常通过 softmax 函数取得).

为什么使用 Cross-entropy

1. **概率解释** Cross-entropy 提供了真实概率分布和预测概率分布之间的差异的度量. 较低的交叉熵值表明模型的预测与真实标签比较相似.
2. **可微分** 对于学习算法, 尤其是神经网络, 具有一个可微分的损失函数是至关重要的. 交叉熵可微分, 使得他可以被用于像梯度下降这样的基于梯度的优化方法.
3. **惩罚自信的错误预测** 交叉熵作为 loss 函数的一个优点是它如何处理预测. 如果一个模型对一个错误的类别预测 0.99 的概率, 对正确的类别预测 0.01 的概率, 交叉熵会变得非常大. 这个特征确保了当模型非常自信得犯错的时候, 会被严重惩罚;
4. **和 Softmax 配合良好**交叉熵损失结合输出层中的 softmax 激活函数是分类问题中流行的组合. 从这个组合中获得的梯度有良好的属性, 使得模型可以更快收敛, 并且经常相较于其他的激活函数-损失函数对具有更好的解决方案.
5. **解决饱和问题** 在神经网络中, sigmoid 激活函数和 quadratic cost 可以导致 ‘‘饱和’’ 问题, 此时神经元的输出和梯度都几乎是 0, 使得网络难以训练. 交叉熵不会受这个问题困扰, 使得训练更加高效.

模型训练

神经网络模型的结构确定后，下一个关键步骤就是在数据上对其进行训练.

1. **编译模型**
   - **损失函数** 模型使用 'categorical_crossentropy' 作为损失函数.
   - **优化器** 使用 'Adam' 作为优化器. Adam 对每个参数调节学习率. 其具有高效率与低内存需求. 'learning_rate' 战术决定了优化器最小化 loss 的步长, 'decay' 可以随时间降低学习率来允许在之后的训练过程中更细化的权重更新.
   - **矩阵** 'accuracy' 是在训练过程中监控的矩阵. 准确率提供了一个清晰的, 直观的对模型执行情况的理解: 表示正确分类的实例在总实例中占的百分比.

模型训练

神经网络模型的结构确定后，下一个关键步骤就是在数据上对其进行训练.

1. **编译模型**
2. **拟合模型**
   - **数据源** 模型借助 'fit_generator' 方法进行训练，该方法允许边读取边数据增强 (data augmentation on-the-fly)，且更加内存高效. 该方法当训练集太大而无法整个放入内存的时候更合适.
   - **轮数** 模型训练了 50 轮. 每轮代表一次对于所有训练样本的前向传播和反向传播.
   - **Batch Size** 每一轮的步数 ('28709 // 64') 表示数据每次被传入大小为 64 个实例的批次. 类似的，对于验证阶段，使用批次大小 '7178 // 64'. 使用批次加速了训练过程，因为模型权重的更新是在每一轮后进行的，而不是在每个数据点之后.
   - **验证数据** 模型的表现在一个单独的验证数据几上并行验证. 这帮助了监控任何的过拟合信号，当模型在训练数据上表现异常良好，而对于新的没有见过的数据非常糟糕的时候就发生了过拟合.

使用机器学习方法预测从泰坦尼克灾难中生存的可能

使用逻辑回归模型的贷款预测

借助 scikit-learn 的逻辑回归模型对是否可以给用户贷款进行预测. 该实验主要内容在于数据预处理.

模型训练和预测

```
LR=LogisticRegression()
LR.fit(X_train,y_train)
y_hat=LR.predict(X_test)

# prediction summary by species
print(classification_report(y_test,y_hat))

# accuracy score
LR_SC=accuracy_score(y_hat,y_test)
print('accuracy is',accuracy_score(y_hat,y_test))
```

逻辑回归

'Logistic Regression' 是一个用来进行二元分类问题的统计方法，一个分类问题指预测一个特定数据属于两个中的哪一个类．被叫做 "logistic" 回归因为是基于逻辑函数的，这被用于建模将一个给定的输入点划入两个分类之一的概率．

训练模型

LR=LogisticRegression()
LR.fit(X_train,y_train)

- 首先，借助‘LogisticRegression()’来获取一个逻辑回归模型.
- 下一步，‘fit’方法将模型在训练数据上进行训练. 这里，‘X_train’是具有特征的训练数据，‘y_train’是训练数据对应的标签.

进行预测

y_hat=LR.predict(X_test)

在将模型在训练数据上训练后，下一步就是在新的没有见过的模型上进行预测. ‘predict’ 方法将测试数据 ‘X_test’ 作为参数，返回预测标签 ‘y_hat’.

分类报告

print(classification_report(y_test,y_hat))

'classification_report' 提供了模型表现的准确度，召回度和 F1-score.

- **Precision 准确度**在所有预测为某个特定类的例子中，多少是真阳性;
- **Recall 召回率**在所有属于一个特定类的实例中，哪些被正确预测了，即真阳性与真阴性的和;
- **F1-score** 准确度和召回率的调和平均值，提供了两个之间的平衡指标.

Precision 准确度

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{2}$$

当假阳性的代价高的时候, 准确度是关键的.

Recall 召回率

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{3}$$

当假阴性的代价高的时候, 召回率是重要的.

准确率分数

```
LR_SC=accuracy_score(y_hat,y_test)
print('accuracy is',accuracy_score(y_hat,y_test))
```

准确率分数是一个可以表明在测试数据中正确预测的分类的占比的量. 计算方式为将正确预测的数量除以预测的总数.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{4}$$

对于本处的贷款预测:

- 如果一个贷款请求倾向于被允许, 模型会预测 '1', 否则为 0.
- 准确率通过将预测结果与实际情况对比告诉我们多少的贷款请求被我们的模型正确预测.

模型局限

逻辑回归具有这样的局限:

线性决策边界 逻辑回归假设一个线性决策边界，这可能不能捕获数据中的复杂关系.

特征的独立性 逻辑回归假设输入特征是独立的，意味着他们不会互相影响. 如果一些特征是强相关的，这可能影响模型的参数.

使用机器学习方法预测从泰坦尼克灾难中生存的可能

在 Boston Hosting 数据集上进行房价预测

借助 PyTorch 搭建全连接模型对 Boston Hosting 数据集进行房价预测.

模型结构

```
    # boston housing model
class BostonHousingModel(nn.Module):
    def __init__(self):
        super(BostonHousingModel,self).__init__()
        self.layer1=nn.Linear(13,64)
        self.layer2=nn.Linear(64,64)
        self.layer3=nn.Linear(64,1)

    def forward(self,x):
        x=torch.relu(self.layer1(x))
        x=torch.relu(self.layer2(x))
        x=self.layer3(x)
        return x
```

模型借助 PyTorch 框架, 使用 'nn.Module' 基类来定义神经网络结构.

模型初始化

第一层

全连接层，输入特征为 13，对应 Boston Housing dataset 中的 13 个特征，输出特征数量为 64.

第二层

全连接层，输入特征数量为 64，输出特征数量为 64.

第三层

全连接层，输入特征数量为 64，输出特征数量为 1，即表示预测房价结果.

前向传播

前向传播方法描述了输入数据是如何流过模型的.

- **第一步** 将输入 'x' 通过 'layer1', 应用 ReLU(Rectified Linear Unit) 激活函数. ReLu 函数引入非线性, 允许模型从错误中学习和调整, 对于学习复杂的特征至关重要;
- **第二步** 将前一步的输出传入 'layer2', 再次使用 ReLU 激活函数;
- **最后一步** 将结果传过 'layer3'. 在这一层没有激活函数, 因为这是一个回归问题, 模型被设计为直接输出连续值.

模型训练

- **损失函数** 使用 'nn.MSELoss()' 作为损失函数. 平均平方差损失计算了预测和实际值的平均平方差, 对于房价预测问题是合适的选择.
- **优化器选择** 选择 Adam 优化器来调节模型参数. 学习率为 '0.001'.
- **训练轮数** 训练 500 轮, 并记录日志.

改进模型

在前面基础模型的基础上，使用 Dropout 进行改进．在训练策略上，使用学习率规划下降和提前停止来优化训练．

```
class OptiBostonHousingModel(nn.Module):
    def __init__(self):
        super(OptiBostonHousingModel,self).__init__()
        self.layer1=nn.Linear(13,128)
        self.layer2=nn.Linear(128,64)
        self.dropout=nn.Dropout(0.5)
        self.layer3=nn.Linear(64,1)

    def forward(self,x):
        x=torch.relu(self.layer1(x))
        x=self.dropout(x)
        x=torch.relu(self.layer2(x))
        x=self.dropout(x)
        x=self.layer3(x)
        return x
```

模型结构

仍旧是三层全连接层，但是第二层和第三层之间使用了 Dropout 率为 0.5 的 dropout 层，随机丢弃一半输入数据来帮助避免过拟合.

学习率规划器

使用 'StepLR' 规划器来在训练过程中调节学习率. 特别的，学习率每 100 轮被乘以一个参数 'gamma'(此处为 0.1). 随时间降低学习率可以帮助细化模型的权重，使其更加贴近最优方案.

使用机器学习方法预测从泰坦尼克灾难中生存的可能

CNN 进行 MNIST 手写数字识别

借助 PyTorch 框架, 定义一个 CNN 进行 MNIST 手写数字识别.

模型结构

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        self.conv_layers = nn.Sequential(
            # (n, 1, 28, 28) -> (n,32,28,28)
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            # (n,32,28,28)->(n,32,14,14)
            nn.MaxPool2d(kernel_size=2, stride=2),
            # (n,32,14,14)->(n,64,14,14)
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            # (n,64,14,14)->(n,64,7,7)
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
```

模型结构

```
        self.fc_layers = nn.Sequential(
            # (n,64*7*7)->(n,128)
            nn.Linear(64 * 7 * 7, 128),
            nn.ReLU(),
            # (n,128)->(n,10)
            nn.Linear(128, 10),
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = x.view(x.size(0), -1)  # flatten the tensor
        x = self.fc_layers(x)
        return x
```

模型结构

1. **卷积层 (‘conv_layers’)** 这些层用来处理输入图像的空间信息，检测局部图案，边缘，纹理等．通过级联卷积和池化操作以分层方式转换图像．

   1.1 **第一卷积层**

   - **卷积** 这一层使用单个通道 (因为 MNIST 图片为灰度图片)，将其转化为 32 个通道，借助 3×3 卷积核．大小为 1 的填充确保了输出和输入的宽和高相同，因此输出形状为 $(n, 32, 28, 28)$.
   - **激活函数** ‘ReLU’ 激活函数引入了非线性．
   - **池化** 之后的 ‘MaxPool2d’ 操作使用了步长为 2 的池化核大小为 2×2 的池化操作来降低空间维度大小到原来的一半，使得输出形状为 $(n, 32, 14, 14)$.

   1.2 **第二卷积层**

   - 卷积：将前一层的 32 个通道借助 3×3 的卷积核转化为 64 通道，由于填充为 1，保留了原本的维度．
   - 激活函数：ReLU.
   - 池化：另一个 ‘MaxPool2d’ 操作将其维度降低到原来的一半，输出形状为 $(n, 64, 7, 7)$.

模型结构

2. 全连接层（'fc_layers'）

- 这些层分析卷积层提取和学到的特征，将其分类到 10 个可能的数字类中.

2.1 第一个全连接层

- **扁平化**在将其从卷积层传入全连接层之前，将 3D 张量 $(64, 7, 7)$ 扁平化到 1D 的 3136 个元素的张量.
- **线性变形** 将 3136 个节点降低到 128 个节点.
- **激活函数** 再次使用 'ReLU' 函数来引入非线性.

2.2 第二个全连接层

- 使用了 128 个来自前一层的节点，将其减少到 10 个节点，对应 10 个 MNIST 中可能的数字类 (0 到 9).

模型训练

接下来介绍对模型的训练. 首先将模型和数据移动到 CUDA GPU 上, 然后对模型进行训练.

```
device = torch.device("cuda" if torch.cuda.is_available else "cpu")
# initialize the model
model = CNN().to(device)
```

首先判断 cuda 是否可用, 如果可用则使用其进行加速. 否则在 cpu 上进行训练.

损失和优化器配置

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

- Loss 函数使用交叉熵，对于分类任务这是常用的；
- 使用 Adam 优化器，以及 0.001 的学习率来调节模型参数；

训练循环

```
num_epochs = 10
for epoch in range(num_epochs):  # loop over the dataset multiple
↪    times
    running_loss = 0.0
    model.train()  # set model to train mode
    for inputs, labels in trainloader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        # forward, backward, optimize
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
    print(f"Epoch {epoch+1},loss: {running_loss/len(trainloader)}")
```

训练循环

- 模型进行 10 轮训练;
- 在数据集上迭代之前, ‘running_loss’ 被设置为 0 来累加一轮的损失.
- 借助 ‘model.train()’ 将模型设置到训练模式. 该模式影响 dropout 或者 batch normalization 层这种在训练时和测试时表现不同的层.
- 对来自 ‘trainloader’ 数据的每个批次 (‘inputs’ 和对应的 ‘labels’):
  - 数据被转移到配置的装置 (CUDA 或 CPU);
  - 梯度被清零来确保没有来自前一次迭代的累加;
  - 一个前向过程计算模型的预测 (‘outputs’);
  - loss 被在预测和真实标签之间计算出来;
  - 一个反向传播过程计算 loss 与对应的模型参数的梯度值;
  - 优化器基于计算的梯度更新模型的参数;
  - 该批次的 loss 被累加到 ‘running_loss’;
- 在所有的批次被处理后, 该轮次的平均 loss 被打印.

训练循环的验证阶段

```
...
model.eval()  # set the model to evaluation mode
with torch.no_grad():
    running_loss = 0.0
    correct_predictions = 0
    total_predictions = 0
    for inputs, labels in validloader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total_predictions += labels.size(0)
        correct_predictions += (predicted == labels).sum().item()
    print(
        f"Validation loss: {running_loss/len(validloader)}, Validation
        ↪  accuracy: {correct_predictions/total_predictions*100}%"
    )
```

训练循环的验证阶段

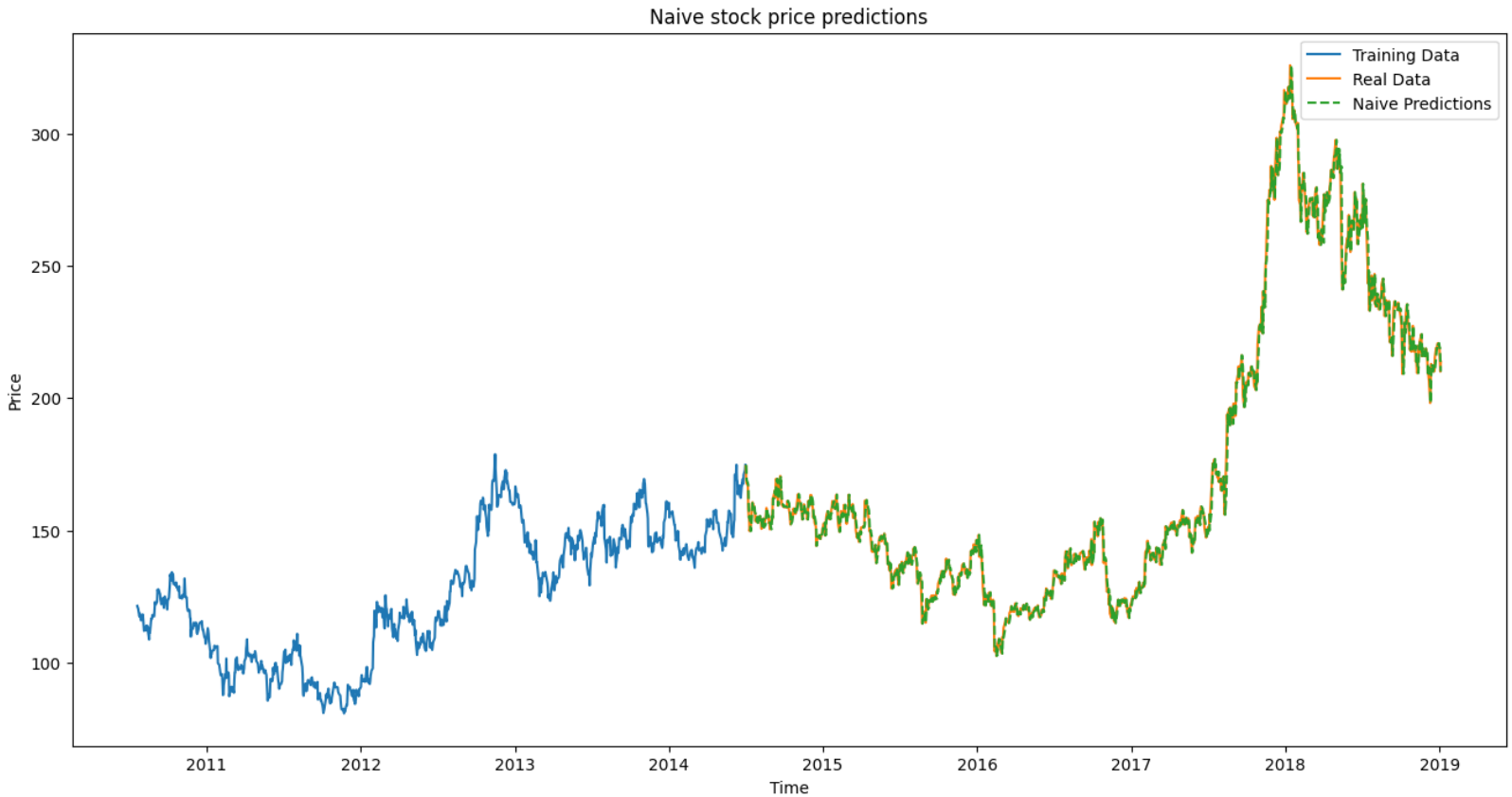
- 模型被借助 ‘model.eval()’ 设置为验证模式. 这确保了 dropout 和 batch normalization 等层工作在验证模式;
- ‘torch.no_grad()’ 确保了在这一阶段不会计算梯度, 减少了内存消耗;
- 对来自 ‘validloader’ 的每一批次数据:
  - 数据被转移到 device 上;
  - 一个前向传播过程计算了模型预测;
  - loss 被计算并添加到 ‘running_loss’;
  - 预测值被与真实标签比较来确定准确度.
- 在所有的批次被处理完之后, 平均的验证损失和准确率都被打印;

使用机器学习方法预测从泰坦尼克灾难中生存的可能

股票价格预测

使用 LSTM 借助历史收盘股价进行股票价格预测. 实际上如果想要作为投资参考, 对股票的收益率或者涨跌进行预测是更好的方式, 对股票价格进行预测仅仅用来学习.

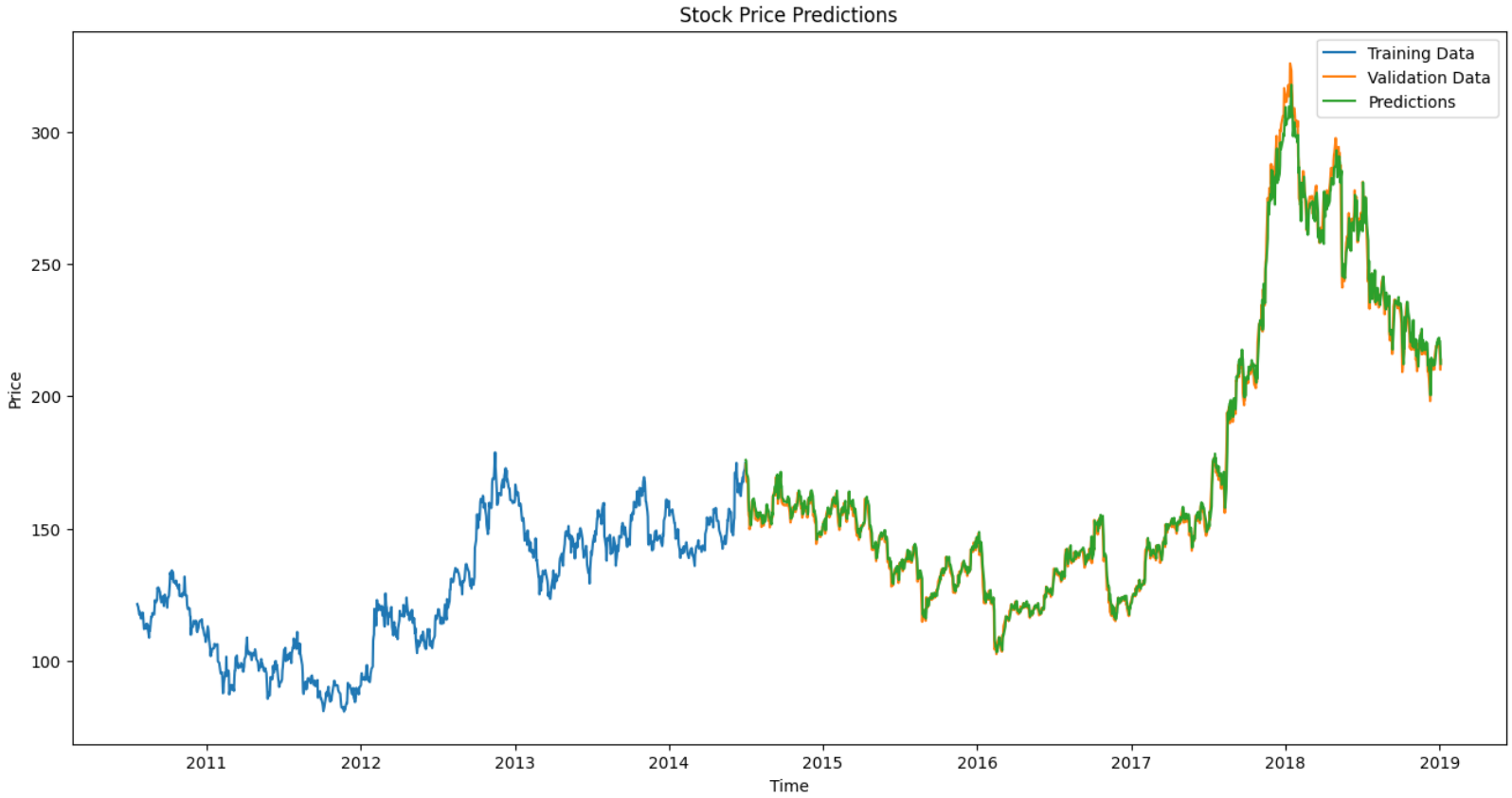
Figure: 如图, 仅仅使用昨天的数据作为今天测预测值也可以获得看起来良好的预测结果, 但是实际上是没有用的



股票价格预测

使用 LSTM 借助历史收盘股价进行股票价格预测. 实际上如果想要作为投资参考, 对股票的收益率或者涨跌进行预测是更好的方式, 对股票价格进行预测仅仅用来学习.

Figure: 如图为 LSTM 的预测结果



模型结构

```
lstm_model = Sequential()
lstm_model.add(
    LSTM(units=50, return_sequences=True,
    ↪    input_shape=(X_train.shape[1], 1))
)
lstm_model.add(Dropout(0.2))
lstm_model.add(LSTM(units=50))
lstm_model.add(Dropout(0.2))
lstm_model.add(Dense(1))

inputs = new_dataset[len(new_dataset) - len(valid_data) - 60 :].values
inputs = inputs.reshape(-1, 1)
inputs = scaler.transform(inputs)
print(inputs.shape)
```

模型结构

```
lstm_model.compile(loss="mean_squared_error", optimizer="adam")

# patience is the number of epochs to check for improvement
early_stop=EarlyStopping(monitor='val_loss', patience=10,
    verbose=1)

lstm_model.fit(X_train, y_train, epochs=100, batch_size=1, verbose=2,
    validation_split=0.2, callbacks=[early_stop])
```

模型结构

在本实验 vs, 借助 Keras 搭建了一个 LSTM 神经网络. LSTM 是为顺序数据特别设计的, 对于时序数据有良好预测性能, 且相比于 RNN 有效解决了长期记忆衰退问题的模型. 对其结构介绍如下.

输入层

模型期望输入数据有形状*(X_train.shape[1], 1)*. 这意味着每个训练样本应该是一个具有 *X_train.shape[1]*步的序列, 且在每一步只有一个特征.

模型结构

第一个 LSTM 层

- 有 50 个神经元;
- 在 tensorflow 中, 设置 ***return_sequences=True*** 意味着这个 LSTM 层会将整个序列传递给下一层, 对于堆栈的 LSTM 层是必要的.

模型结构

关于 *return_sequences*

LSTM 层中的 ***return_sequences*** 控制了 LSTM 输出的是序列还是整个序列的最后一个输出.

设置 ***return_sequences*** 为 'True' 或者 'False' 有特定的使用场景:

1. ***return_sequences=True***
   - LSTM 层会为输入序列中的每个时间步都产生一个输出, 意味着输出会和输入有相同的长度;
   - 将 LSTM 层堆叠的场景下这是必须的. 因为下一层需要和上一层期望相同长度的序列作为输入.
   - 如果进行的是 seq2seq 的预测, 则也应该使用这种方式.
2. ***return_sequences=False***
   - LSTM 层会对整个输入序列产生一个单独的输出值. 这个只对应了最后一个时间步;
   - 当仅仅对于最后的预测结果感兴趣的时候, 这是合适的. 在许多 seq2val 的预测中采用这种方式.
   - 在一个全连接层之前, 通常也这样使用, 尤其是在分类或回归任务的模型中.

模型结构

第一个 Dropout 层

该层随机设置 20% 的输入单元为 0, 这可以帮助避免过拟合.

为什么在 LSTM 层之间使用 Dropout 层

- Dropout 层放在 LSTM 层之间的时候, 即对 LSTM 层的输出进行了 dropout, 在训练时对其中的部分值随机设置为 0.
- Dropout 可以有效避免过拟合, 尤其是在较小的数据集情况下 (如本实验的数据集).
- 如果模型本身没有过拟合, 添加 dropout 不仅可能不会提供帮助, 反而可能降低模型的表现. 因此, 需要监控验证表现并进行及时的调整.

模型结构

使用 Dropout 可能的缺点

- 在 RNN 中使用 dropout, 尤其是通过 ***recurrent_dropout***, 会延长训练时间, 因为在 Keras 中对于 ***recurrent_dropout*** 的实现关闭了对于 LSTM 层的一些性能优化方法;
- 可能引入了过多的 dropout, 这可能降低模型的学习能力. 在规范化模型和允许他从数据中学习之间应该取得一个平衡.

模型结构

第二个 LSTM 层

- 具有 50 个单元;
- 因为后面没有后继的 LSTM 层, 因此 ***return_sequences*** 没有设置, 默认为 False. 这意味着该层只会输出最后一个时间步的值.

第二个 Dropout 层

仍旧 20%dropout 率的 Dropout 层, 用来避免过拟合;

输出层

全连接层, 仅有一个神经元. 因为这里没有设置激活函数, 默认是线性激活函数, 使其适合回归任务.

模型编译和训练

编译

- 模型使用平均平方差 loss 进行编译，这对于回归任务是常用的选择．使用的优化器是 'Adam'，对于许多深度学习任务这都是一个流行且高效的优化器．

提前停止回调

该回调函数监控了验证损失 (val_loss)．如果在 10 轮训练中验证 loss 没有提升 $(patience=10)$，训练会停止．这可以帮助避免过拟合，并且如果模型已经收敛，可以帮助减少训练时间;

模型训练

- 模型训练了最多 100 轮，批次大小为 1．意味着模型在每个样本之后都会更新参数．
- 20% 的数据用作验证数据．$(validation\_split=0.2)$．
- 使用了提前停止回调函数．

使用机器学习方法预测从泰坦尼克灾难中生存的可能

泰坦尼克

根据穿上人员的性别，年龄，职业等信息，设计一种算法来预测泰坦尼克号上乘客的生存概率．借助该问题，学习使用‘scikit-learn’库和 xgboost 等库的各种机器学习方法，例如 SVM 支持向量机，决策树，线性模型，朴素贝叶斯，高斯过程和决策树等．

使用的机器学习模型介绍

```
    # machine learning algorithms (MLA) selection and Initialization
MLA = [
    # ensemble methods
    ensemble.AdaBoostClassifier(),
    ensemble.BaggingClassifier(),
    ensemble.ExtraTreesClassifier(),
    ensemble.GradientBoostingClassifier(),
    ensemble.RandomForestClassifier(),
    # Gaussian Processes
    gaussian_process.GaussianProcessClassifier(),
    # GLM
    linear_model.LogisticRegressionCV(),
    linear_model.PassiveAggressiveClassifier(),
    linear_model.RidgeClassifierCV(),
    linear_model.SGDClassifier(),
    linear_model.Perceptron(),
```

使用的机器学习模型介绍

```
    # Naive Bayes
    naive_bayes.BernoulliNB(),
    naive_bayes.GaussianNB(),
    # Nearest Neighbor
    neighbors.KNeighborsClassifier(),
    # SVM
    svm.SVC(probability=True),
    svm.NuSVC(probability=True),
    svm.LinearSVC(),
    # Trees
    tree.DecisionTreeClassifier(),
    tree.ExtraTreeClassifier(),
    # Discriminant Analysis
    discriminant_analysis.LinearDiscriminantAnalysis(),
    discriminant_analysis.QuadraticDiscriminantAnalysis(),
    # XGBoost
    XGBClassifier(),
]
```

使用的机器学习模型介绍

集成方法 (Ensmble methods): 这些方法通常集合了多个模型 (常常称为 ‘‘基础学习者”) 来提升泛化能力和鲁棒性

- AdaBoostClassifier: 一个拟合一系列的弱学习者 (通常是决策树) 的自适应增强算法. 每棵决策树都会纠正其前辈的错误.
- BaggingClassifier: 使用 bagging (Bootstrap Aggregating) 来在数据集的随机子集上拟合一个分类器的多个实例. 子集是通过替换获取的.
- ExtraTreesClassifier: 代表 “Extremely Randomized Trees”. 像一个更加随机版本的随机森林, 其中最好的划分从一个随机的子集和阈值中选取;
- GradientBoostingClassifier: 顺序添加预测器来纠正由前面的预测器做出的错误. 和 AdaBoost 不同, 该分类器专注于直接减少残差.
- RandomForestClassifier: 在训练中建立一个决策树 ‘森林’. 每棵树都在数据和特征的一个随机子集上训练. 预测被整合 (对于回归任务取平均, 对于分类任务取多数投票).

使用的机器学习模型介绍

高斯过程

- GaussianProcessClassifier：一个用于回归和概率分类的非参数贝叶斯方法．他在函数上定义一个分布，借助核技巧在高维空间中进行操作．

泛化线性模型

- LogisticRegressionCV：具有用于自动参数调整的集成交叉验证的逻辑回归 (用于二元分类)．
- PassiveAggressiveClassifier：一种适合大规模学习的在线学习算法，被称为 ‘‘被动攻击”，因为在正确分类时保持被动，并在发生错误时变得攻击．
- RidgeClassifierCV：使用 L2 正则化的线性分类器 (对系数的平方幅度添加惩罚)．该版本包含集成的交叉验证．
- SGDClassifier：线性分类器 (像 SVM 和逻辑回归)，具有随机梯度下降 (SGD) 训练的线性分类器．
- Perceptron：线性分类器的一种，是一种监督二元分类算法．

使用的机器学习模型介绍

朴素贝叶斯

基于应用贝叶斯理论以及特征之间的强独立性假设.

- BernoulliNB: 二元数据使用. 假设特征是二元的.
- GaussianNB: 假设特征有高斯分布.

最近邻居

- KNeighborsClassifier: 基于训练集中他的 'k' 个最近的邻居来分类一个样本.

支持向量机 (SVM)

旨在找到最能将数据集划分为类别的超平面;

- SVC: 使用核技巧来变形输入数据, 然后识别最优的超平面;
- NuSVC: 和 SVC 类似, 但是使用一个参数 'nu' 来控制支持向量的数量.
- LinearSVC: 不是用核技巧的线性 SVM.

使用的机器学习模型介绍

决策树

- DecisionTreeClassifier：构造一棵树，其中每个节点都根据特征阈值作出决策．分割是根据熵或者基尼杂质 (Gini impurity) 等依据做出的；
- ExtraTreeClassifier：类似决策树，但是划分的更加随机．

判别分析：用于分类和降维

- LinearDiscriminantAnalysis，线性判别分析 (LDA)：假设每个类具有高斯分布，并共享相同的协方差矩阵；
- QuadraticDiscriminantAnalysis，二次判别分析 (QDA)：与 LDA 类似，但是允许每个类都有其协方差矩阵；

XGBoost

- XGBClassifier：优化的梯度增强库．代表 eXtreme Gradient Boosting．以其速度和性能闻名．

使用的机器学习模型介绍

```
cv_split = model_selection.ShuffleSplit(
    n_splits=10, test_size=0.3, train_size=0.6, random_state=0
)
# create table to compare MLA metrics
MLA_columns = [
    "MLA Name",
    "MLA Paramaters",
    "MLA Train Accuracy Mean",
    "MLA Test Accuracy Mean",
    "MLA Test Accuracy 3*STD",
    "MLA Time",
]
MLA_compare = pd.DataFrame(columns=MLA_columns)
# create table to compare MLA predictions
MLA_predict = data1[Target]
```

训练并进行预测

```
# index through MLA and save performance to table
row_index = 0
for alg in MLA:
    # set name and parameters
    MLA_name = alg.__class__.__name__
    MLA_compare.loc[row_index, "MLA Name"] = MLA_name
    MLA_compare.loc[row_index, "MLA Parameters"] =
     ↪    str(alg.get_params())

    # score model with cross validation
    cv_results = model_selection.cross_validate(
        alg,
        data1[data1_x_bin],
        data1[Target],
        cv=cv_split,
        return_train_score=True,
    )
```

训练并进行预测

```
MLA_compare.loc[row_index, "MLA Time"] =
    cv_results["fit_time"].mean()
MLA_compare.loc[row_index, "MLA Train Accuracy Mean"] =
    cv_results[
  "train_score"
].mean()
MLA_compare.loc[row_index, "MLA Test Accuracy Mean"] =
    cv_results[
  "test_score"
].mean()
# if this is a non-bias random sample, then +/-3 standard
    deviations(std)
# from the mean, should statistically capture 99.7% of the subsets
MLA_compare.loc[row_index, "MLA Test Accuracy 3*STD"] = (
   cv_results["test_score"].std() * 3
)  # the worst that can happen

# save MLA predictions
```

训练并进行预测

交叉验证划分

借助 'ShuffleSplit' 方法订一个 'cv_split' 这个交叉验证策略. 策略创建 10 个划分, 将数据中的 60% 用于训练, 30% 用于测试. 该方法不是用整个数据集, 因为每个划分中有 10% 的数据没有被使用.

比较表

两个 Pandas DataFrames('MLA_conpare' 和 'MLA_predict') 被初始化来存储每个算法的性能指标和预测.

算法循环

在 'MLA' 列表中的每个算法都被循环经过, 不同的任务被执行:

- 算法名字和参数被保存;
- 算法在数据集上使用前面提到的划分策略进行交叉验证. 计算并存储训练准确度, 测试准确度, 拟合时间等指标.
- 算法在数据的一个子集上进行训练, 在同一个子集上进行预测. 这些预测会被保存, 在之后可视化训练结果的时候使用.

结果