

期末实验报告



项 目 名 称 : ANN 机器学习算法应用案例

学 生 姓 名 : 刘喆恺

学 号 : 10220712

专 业 名 称 : 数学与应用数学（强基计划）

所 在 学 院 : 数学学院

日 期 : 2024 年 11 月 28 日

ANN 机器学习算法应用案例

一、简介

本文中，使用 Python 搭建了一个神经网络，来处理分类任务。项目核心在于仅使用 Numpy 库，纯手工搭建了一个可以自由选择神经网络层数和每一层的神经元个数，使用 Adam 优化算法，并可以实现自动求导（反向传播）的功能，避免使用 Pytorch 库。

在本文实验中，有两个 hidden layer，每个层有 64 个 neurons，分三个类（input=2, output=3）100 个数据，然后建立了一个 training set 和 testing set 没有 validation set。以下为全部实验代码，并穿插代码与算法解释。最后将展示实验结果。全部代码请参考（SGD/Adagrad/PMSprop/Adam）：

<https://github.com/Zhekai-Liu/Manual-Neural-network.git>

二、实验过程

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

nnfs.init()
```

首先引入 Numpy 库函数，并引入 nnfs.init() 函数，初始化环境，方便实验后续的复现。

```
class Lay_dense:
    def __init__(self, n_inputs, n_neurons,
weight_regularizer_l1=0., weight_regularizer_l2=0.,
bias_regularizer_l1=0., bias_regularizer_l2=0.):
```

定义神经层，并定义初始化，分别是输入，神经元个数，正则化参数作为惩罚项，防止过拟合。

```
self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
self.biases = np.zeros((1, n_neurons))
```

初始化神经元的权重和偏度。

```
self.weight_regularizer_l1 = weight_regularizer_l1
self.weight_regularizer_l2 = weight_regularizer_l2
self.bias_regularizer_l1 = bias_regularizer_l1
self.bias_regularizer_l2 = bias_regularizer_l2
```

正则化部分，把正则化参数引入，系数用来刻画惩罚项强度。

```
def forward(self, inputs):
    self.inputs = inputs
    self.output = np.dot(inputs, self.weights) + self.biases
```

向前传播，形如 $\mathbf{wx} + \mathbf{b}$ 。

```
def backward(self, dvalues):
    self.dweights = np.dot(self.inputs.T, dvalues)
    self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
```

反向传播：对于权重，他的梯度就是 **inputs** 的系数，对于偏度，其实就是对于输入值梯度的求和。原因如下图：

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{n_{\text{samples}}} \frac{\partial L}{\partial y_i} \cdot \frac{\partial y_i}{\partial b}$$

由于 $\frac{\partial y}{\partial b} = 1$ ，公式简化为：

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{n_{\text{samples}}} \frac{\partial L}{\partial y_i}$$

```
if self.weight_regularizer_l1 > 0:
    dL1 = np.ones_like(self.weights)
    dL1[self.weights < 0] = -1
    self.dweights += self.weight_regularizer_l1 * dL1
if self.weight_regularizer_l2 > 0:
    self.dweights +=
```

```

2*self.weight_regularizer_l2*self.weights
    if self.bias_regularizer_l1 > 0:
        dL1 = np.ones_like(self.biases)
        dL1[self.biases < 0] = -1
        self.dbiases += self.bias_regularizer_l1 * dL1
    if self.bias_regularizer_l2 > 0:
        self.dbiases +=
2*self.bias_regularizer_l2*self.biases

```

因为我们引入了正则化项，因此在计算损失函数梯度时，也需要考虑到正则化部分的影响，最后加在原梯度之上，构成全梯度。如图（以 L1 正则化为例）：

其中：

- w 是权重向量。
- λ 是正则化强度参数，代码中的 `self.weight_regularizer_l1`。
- 目标是在优化过程中最小化损失函数：

$$\text{Loss} = \text{Original Loss} + \text{L1 Regularization Loss}$$

在反向传播中，我们需要计算正则化项对权重的梯度，公式为：

$$\frac{\partial(\lambda \sum |w|)}{\partial w} = \lambda \cdot \text{sign}(w)$$

其中， $\text{sign}(w)$ 是符号函数：

- 如果 $w > 0$, $\text{sign}(w) = 1$
- 如果 $w < 0$, $\text{sign}(w) = -1$
- 如果 $w = 0$, $\text{sign}(w)$ 通常取为 0

```
self.dinputs = np.dot(dvalues, self.weights.T)
```

`dvalues` 是后端的导数，`inputs` 的倒数是 `weights`，二者相乘，成为 `dinputs`，向前传播。（这个就是向前传播的实现）

```

class Activation_ReLU:
    def forward(self, inputs):
        self.inputs = inputs
        self.output = np.maximum(0, inputs)

```

ReLU 激活函数用于使神经网络变为非线性模型，从而逼近更复杂的函数。

```
def backward(self, dvalues):
    self.dinputs = dvalues.copy()
    self.dinputs[self.inputs <= 0] = 0
```

反向传播需要认为定义导数，因为其不可导，如图：

Recall that the derivative of *ReLU()* with respect to its input is 1, if the input is greater than 0, and 0 otherwise:

$$f(x) = \max(x, 0) \rightarrow \frac{d}{dx}f(x) = 1(x > 0)$$

We can write that in Python as:

```
relu_dz = (1. if z > 0 else 0.)
```

```
class Activation_Softmax:
```

```
    def forward(self, inputs):
        self.inputs = inputs
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
keepdims=True))
        probabilities = exp_values / np.sum(exp_values, axis=1,
keepdims=True)
        self.output = probabilities
```

Softmax 类在实验中用于最后一层，得到各个 **output** 值时，计算 **softmax** 即概率分部，最后取 **argmax** 得到最终结果，但事实上，后文会讲到，在训练过程中，我们并不会去取 **argmax**（因为不可导），而是利用例如交叉熵的算法来替代。（交叉熵实际上来源于信息熵，但是我已经忘掉了，只记得公式了。。。）

```
class Optimizer_Adam:
```

```
    def __init__(self, learning_rate=0.001, decay=0.,
epsilon=1e-7, beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
```

```

self.decay = decay
self.iterations = 0
self.epsilon = epsilon
self.beta_1 = beta_1
self.beta_2 = beta_2

```

Adam 算法事实上和 momentum 算法区别不大（其实 Adam 全称就是 Adaptive Momentum），都是在迭代过程中改进学习率，在算法初期用大学习率来加快收敛速度，在后期降低学习率（decay）从而保证可以下降到极小值点。

```

def pre_update_params(self):
    if self.decay:
        self.current_learning_rate = self.learning_rate * (1.
/ (1. + self.decay * self.iterations))

```

观察`(1. + self.decay * self.iterations)`，分母是不断增大的，并以某一个 decay 率。因此学习率在减小。

公式：

$$\text{current_learning_rate} = \text{learning_rate} \times \frac{1}{1 + \text{decay} \times \text{iterations}}$$

- 学习率会随着迭代次数增加而逐渐减小，防止训练后期跳动过大。

```

def update_params(self, layer):

    if not hasattr(layer, 'weight_cache'):
        layer.weight_momentums = np.zeros_like(layer.weights)
        layer.weight_cache = np.zeros_like(layer.weights)
        layer.bias_momentums = np.zeros_like(layer.biases)
        layer.bias_cache = np.zeros_like(layer.biases)

```

创建这些位置。

```

        layer.weight_momentums = self.beta_1 *
layer.weight_momentums + (1 - self.beta_1) * layer.dweights
        layer.bias_momentums = self.beta_1 * layer.bias_momentums
+ (1 - self.beta_1) * layer.dbiases

```

以上是动量更新部分：

1. 动量更新公式：

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

- m_t ：当前的动量。
- g_t ：当前梯度（`layer.dweights` 和 `layer.dbiases`）。
- β_1 ：动量的指数加权衰减率。

2. 意义：动量可以平滑梯度的更新，减少随机梯度方向的波动。

```
weight_momentums_corrected = layer.weight_momentums / (1 - self.beta_1 ** (self.iterations + 1))  
bias_momentums_corrected = layer.bias_momentums / (1 - self.beta_1 ** (self.iterations + 1))
```

动量偏差修正：

1. 偏差修正公式：

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

- Adam 初始化时，动量会偏向于零，需要用偏差修正因子修正。

2. 意义：消除动量的初始偏差，使更新更加准确。

```
layer.weight_cache = self.beta_2 * layer.weight_cache + (1 - self.beta_2) * layer.dweights ** 2  
layer.bias_cache = self.beta_2 * layer.bias_cache + (1 - self.beta_2) * layer.dbiases**2
```

下一步将要使用缓存值来进行对于 **weight** 和 **bias** 梯度的约束。

1. 缓存更新公式：

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

- v_t ：当前的缓存值。
- g_t ：当前梯度平方。
- β_2 ：缓存的指数加权衰减率。

2. 意义：缓存用于估计梯度的尺度，类似于 RMSprop 方法。

RMSprop 方法请参见 [github](#)。

```
weight_cache_corrected = layer.weight_cache / (1 -  
self.beta_2 ** (self.iterations + 1))  
bias_cache_corrected = layer.bias_cache / (1 - self.beta_2  
** (self.iterations + 1))
```

对于刚刚的缓存，我们也需要做一次修正。其实很简单，就是别让他最开始的时候太小了。

1. 偏差修正公式：

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- 修正缓存的偏差，与动量类似。

2. 意义：保证缓存的估计值更准确。

```
layer.weights += -self.current_learning_rate *  
weight_momentums_corrected / (np.sqrt(weight_cache_corrected) +  
self.epsilon)  
layer.biases += -self.current_learning_rate *  
bias_momentums_corrected / (np.sqrt(bias_cache_corrected) +  
self.epsilon)
```

最后，进行梯度的更新，公式如图：

1. 更新公式：

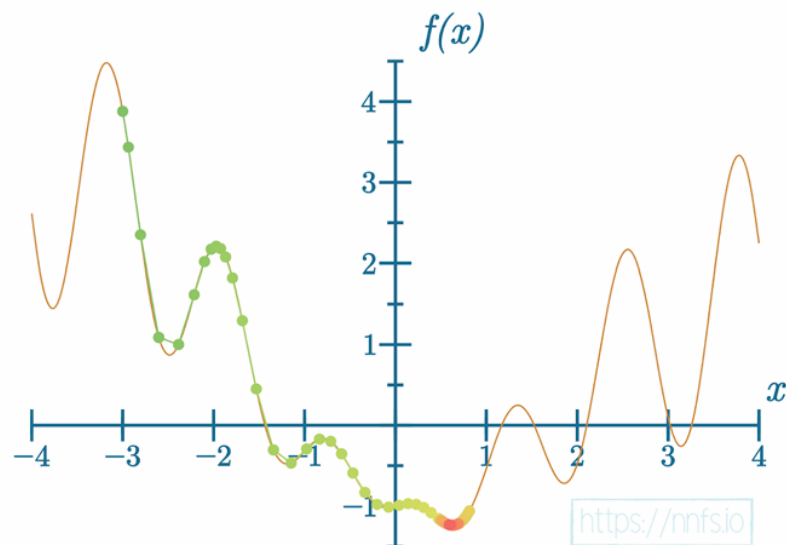
$$w_t = w_{t-1} - \frac{\text{learning rate} \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

- \hat{m}_t ：动量修正值。
- \hat{v}_t ：缓存修正值。
- ϵ ：避免除零。

2. 意义：结合动量和缓存调整参数，既有动量平滑效果，又可以动态调整学习率。

In my perspective, 分母就是来约束或者 generalization 梯度的。学习率 decay 没有什么好说的，最后动量其实就是加权梯度。所有的偏差修正都是为了防止梯度或者梯度范数太小了导致梯度小时或者爆炸而已。事实上，就是之前一次的迭代和现在的梯度做加权，来模拟小球的滚落山下的惯性，来减小陷入

局部最优的可能性。如图：



```
def post_update_params(self):  
    self.iterations += 1
```

更新迭代次数。

事实上，这个算法个人感觉也有缺陷，例如参数太多了。。。。。
调参调起来很麻烦。。。

```
class Loss:
```

```
    def regularization_loss(self, layer):  
        regularization_loss = 0  
        if layer.weight_regularizer_l1 > 0:  
            regularization_loss += layer.weight_regularizer_l1 *  
np.sum(np.abs(layer.weights))  
        if layer.weight_regularizer_l2 > 0:  
            regularization_loss += layer.weight_regularizer_l2 *  
np.sum(layer.weights*layer.weights)  
        if layer.bias_regularizer_l1 > 0:  
            regularization_loss += layer.bias_regularizer_l1 *  
np.sum(np.abs(layer.biases))  
        if layer.bias_regularizer_l2 > 0:
```

```

        regularization_loss += layer.bias_regularizer_l2 *
np.sum(layer.biases*layer.biases)

```

```

    return regularization_loss

```

```

def calculate(self, output, y):
    sample_losses = self.forward(output, y)
    data_loss = np.mean(sample_losses)
    return data_loss

```

这一部分其实就是计算本层神经网络的复杂度，变为惩罚项加在 **loss** 上。小例子：

```

layer = {
    "weights": np.array([0.5, -0.2, 0.1]),
    "biases": np.array([0.1, -0.3]),
    "weight_regularizer_l1": 0.01,
    "weight_regularizer_l2": 0.02,
    "bias_regularizer_l1": 0.01,
    "bias_regularizer_l2": 0.02
}

```

计算过程：

- $L1$ 权重正则化: $0.01 \times (|0.5| + |-0.2| + |0.1|) = 0.01 \times 0.8 = 0.008$
- $L2$ 权重正则化: $0.02 \times (0.5^2 + (-0.2)^2 + 0.1^2) = 0.02 \times 0.3 = 0.006$
- $L1$ 偏置正则化: $0.01 \times (|0.1| + |-0.3|) = 0.01 \times 0.4 = 0.004$
- $L2$ 偏置正则化: $0.02 \times (0.1^2 + (-0.3)^2) = 0.02 \times 0.1 = 0.002$

总正则化损失：

$$\text{Regularization Loss} = 0.008 + 0.006 + 0.004 + 0.002 = 0.02$$

一下我们使用了交叉熵函数，这里就可以注意到了，我们不适用 **argmax**（不可导）而是使用交叉熵。

```

class Loss_cross_entropy(Loss):

```

```

    def forward(self, y_pred, y_true):
        samples = len(y_pred)
        y_pred_clipped = np.clip(y_pred, 1e-7, 1-1e-7)

```

这里的细节是，向前传播前，先做一次 **clip**，防止梯度爆炸。

```
        if len(y_true.shape) == 1:
            correct_confidence = y_pred_clipped[range(samples),
y_true]
        elif len(y_true.shape) == 2:
            correct_confidence = np.sum(y_pred_clipped * y_true,
axis=1)
        negative_log_likelihoods = -np.log(correct_confidence)
```

这里做了一次分裂讨论，其实讨论的就是 **label** 集数据类型格式到底是一维的还是二维的。最后要的结果就是真实 **label** 的概率并进行交叉熵计算，**-log(pi)**。

```
        return negative_log_likelihoods
```

```
class Activation_Softmax_loss():
    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_cross_entropy()
```

最后一层的操作，前文已经解释啦。

```
    def forward(self, inputs, y_true):
        self.activation.forward(inputs) ## to be a probability
distribution
        self.output = self.activation.output # outputs are list of
distributions.
        return self.loss.calculate(self.output, y_true) ##
calculate the cross entropy

    def backward(self, dvalues, y_true):
        samples = len(dvalues)
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)
```

最后一层函数的反向传播。

调整了一下 **label** 的样子，变成以为 **label**。

```

self.dinputs = dvalues.copy()
self.dinputs[range(samples), y_true] -= 1
self.dinputs = self.dinputs / samples

```

这一部分需要解释: `self.dinputs[range(samples), y_true] -= 1`

$$\frac{\partial \text{Loss}}{\partial z_k} = -\frac{\partial}{\partial z_k} \log \left(\frac{e^{z_k}}{\sum_{j=1}^C e^{z_j}} \right)$$

使用链式法则展开:

$$\frac{\partial \text{Loss}}{\partial z_k} = -\frac{1}{\hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial z_k}$$

softmax 对 logits 的导数为:

$$\frac{\partial \hat{y}_k}{\partial z_k} = \hat{y}_k(1 - \hat{y}_k)$$

代入上式:

$$\frac{\partial \text{Loss}}{\partial z_k} = -\frac{1}{\hat{y}_k} \cdot \hat{y}_k(1 - \hat{y}_k)$$

简化后:

$$\frac{\partial \text{Loss}}{\partial z_k} = \hat{y}_k - 1$$

所以其实 softmax 和 cross-entropy 合在一起梯度的计算是很简单的。
神将网络搭建完成。更多内容请参考我的 [github](#)。

实验部分:

```

X, y = spiral_data(samples=100, classes=3)

```

```

dense1 = Lay_dense(2, 64, weight_regularizer_l2=5e-4,
bias_regularizer_l2=5e-4)

```

搭建第一个 layer。64 neurons and 2 inputs.

```
activation1 = Activation_ReLU()
```

建立第一个激活函数。

```
dense2 = Lay_dense(64, 3)
```

第二个 layer, 64 neurons and 3 outputs.

```
loss_activation = Activation_Softmax_loss()
```

Soft-max 激活函数, 生成概率分布。

```
optimizer = Optimizer_Adam(learning_rate=0.02, decay=5e-7)
```

选择优化器, 事实上还有别的优化器, 例如 SGD/Adagrad/PMSprop, 请参见 github。

```
for epoch in range(5001):
```

```
    dense1.forward(X)
```

```
    activation1.forward(dense1.output)
```

```
    dense2.forward(activation1.output)
```

```
    data_loss = loss_activation.forward(dense2.output, y)
```

```
    regularization_loss =  
loss_activation.loss.regularization_loss(dense1) +  
loss_activation.loss.regularization_loss(dense2)
```

```
    loss = data_loss + regularization_loss
```

```
    predictions = np.argmax(loss_activation.output, axis=1)
```

```
    if len(y.shape) == 2:
```

```
        y = np.argmax(y, axis=1)
```

```

accuracy = np.mean(predictions == y)

if not epoch % 50:
    print(f'epoch:{epoch}, ' +
          f'acc:{accuracy:.3f}, ' +
          f'loss:{loss:.3f}(' +
          f'data_loss:{data_loss:.3f},' +
          f'reg_loss:{regularization_loss:.3f}) ' +
          f'lr:{optimizer.current_learning_rate:.5f}')

loss_activation.backward(loss_activation.output, y)
dense2.backward(loss_activation.dinputs)

activation1.backward(dense2.dinputs)
dense1.backward(activation1.dinputs)

optimizer.pre_update_params()
optimizer.update_params(dense1)
optimizer.update_params(dense2)
optimizer.post_update_params()

## test sets
X_test, y_test = spiral_data(samples=100, classes=3)

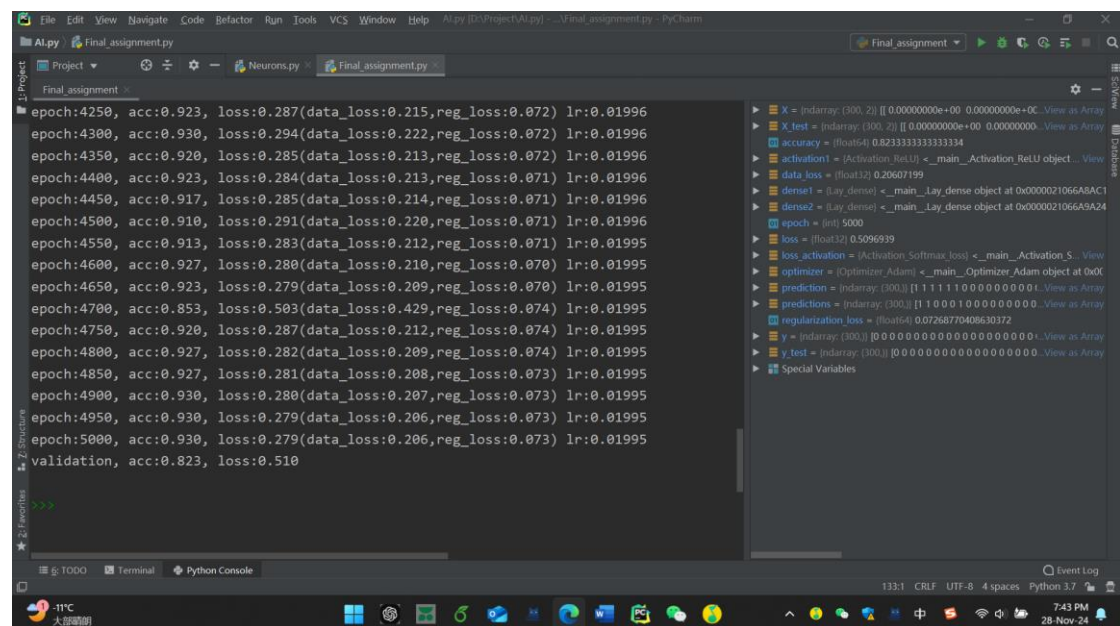
dense1.forward(X_test)
activation1.forward(dense1.output)

dense2.forward(activation1.output)
loss = loss_activation.forward(dense2.output, y)

prediction = np.argmax(loss_activation.output, axis=1)
if len(y_test.shape) == 2:
    y_test = np.argmax(y_test, axis=1)
accuracy = np.mean(prediction == y_test)
print(f'validation, acc:{accuracy:.3f}, loss:{loss:.3f}')

```

实验结果：



```
epoch:4250, acc:0.923, loss:0.287(data_loss:0.215,reg_loss:0.072) lr:0.01996
epoch:4300, acc:0.930, loss:0.294(data_loss:0.222,reg_loss:0.072) lr:0.01996
epoch:4350, acc:0.920, loss:0.285(data_loss:0.213,reg_loss:0.072) lr:0.01996
epoch:4400, acc:0.923, loss:0.284(data_loss:0.213,reg_loss:0.071) lr:0.01996
epoch:4450, acc:0.917, loss:0.285(data_loss:0.214,reg_loss:0.071) lr:0.01996
epoch:4500, acc:0.910, loss:0.291(data_loss:0.220,reg_loss:0.071) lr:0.01996
epoch:4550, acc:0.913, loss:0.283(data_loss:0.212,reg_loss:0.071) lr:0.01995
epoch:4600, acc:0.927, loss:0.280(data_loss:0.210,reg_loss:0.070) lr:0.01995
epoch:4650, acc:0.923, loss:0.279(data_loss:0.209,reg_loss:0.070) lr:0.01995
epoch:4700, acc:0.853, loss:0.503(data_loss:0.429,reg_loss:0.074) lr:0.01995
epoch:4750, acc:0.920, loss:0.287(data_loss:0.212,reg_loss:0.074) lr:0.01995
epoch:4800, acc:0.927, loss:0.282(data_loss:0.209,reg_loss:0.074) lr:0.01995
epoch:4850, acc:0.927, loss:0.281(data_loss:0.208,reg_loss:0.073) lr:0.01995
epoch:4900, acc:0.930, loss:0.280(data_loss:0.207,reg_loss:0.073) lr:0.01995
epoch:4950, acc:0.930, loss:0.279(data_loss:0.206,reg_loss:0.073) lr:0.01995
epoch:5000, acc:0.930, loss:0.279(data_loss:0.206,reg_loss:0.073) lr:0.01995
validation, acc:0.823, loss:0.510
```

Adam 效果图：（不是我跑的，也不是我的代码，但都是 Adam 优化器，同一数据集）

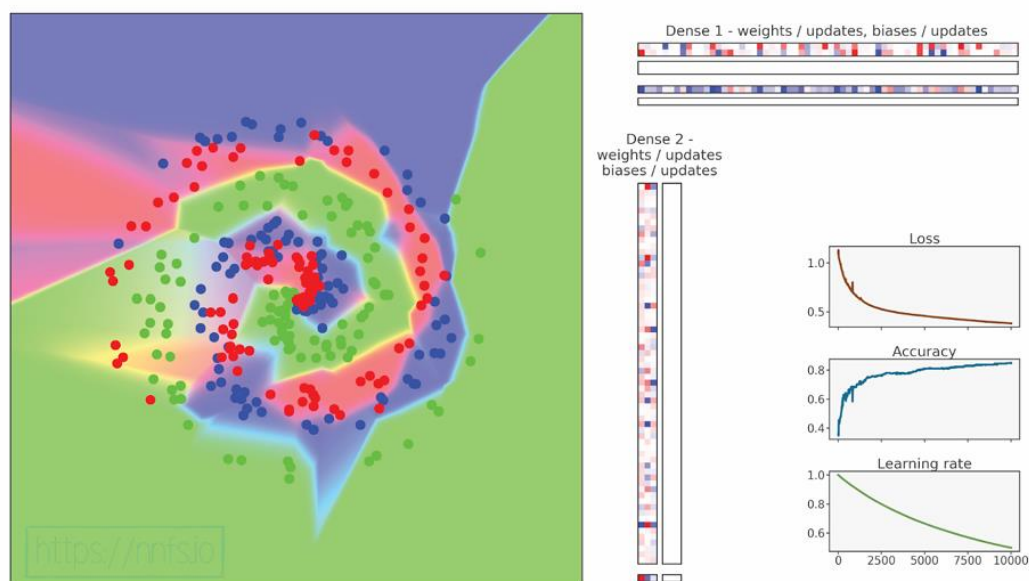


Fig 10.20: Model training with AdaGrad optimizer.

参考文献：

- [1]. <https://github.com/Zhekai-Liu/Manual-Neural-network.git>
- [2]. *Neural Networks from Scratch in Python* Harrison Kinsley & Daniel Kukiela