Project 1

曾晓玮 20307100124

2023 年 4 月 15 日

本项目使用语言为 python，需要的 packages 如下：

[ ]:
```
import numpy as np
from scipy.io import loadmat, savemat
import matplotlib.pyplot as plt
from PIL import Image
from scipy.signal import convolve2d as conv2
```

本项目的实验顺序为：

1. Q0. Baseline
2. Q3. Vectorization
3. Q1. Changing Hidden Units
4. Q6. Introducing Bias
5. Q2. Momentum
6. Extra1. Adam
7. Extra2. Xavier Initialization
8. Q5. Softmax and Cross-Entropy Loss
9. Extra3. Minibatch
10. Q8. Fine Tuning
11. Q4. L2 Regularization and Early Stopping
12. Q7. Dropout
13. Q9. Data Augmentation
14. Extra4. RELU
15. Q10. Convolution2D
16. Final. Parameter Tuning

以上任务的代码被分别放置在文件夹中，主代码文件都为 `example_neuralNetwork.py`，直接运行即可。

Q0. Baseline

我首先将 sample code 中 matlab 语言的代码翻译为 python 语言，然后开始任务。Baseline 代码的错误率在 51.1% 左右，非常高，分类效果差；收敛速度较慢，且运算时间较长，效率并不高。

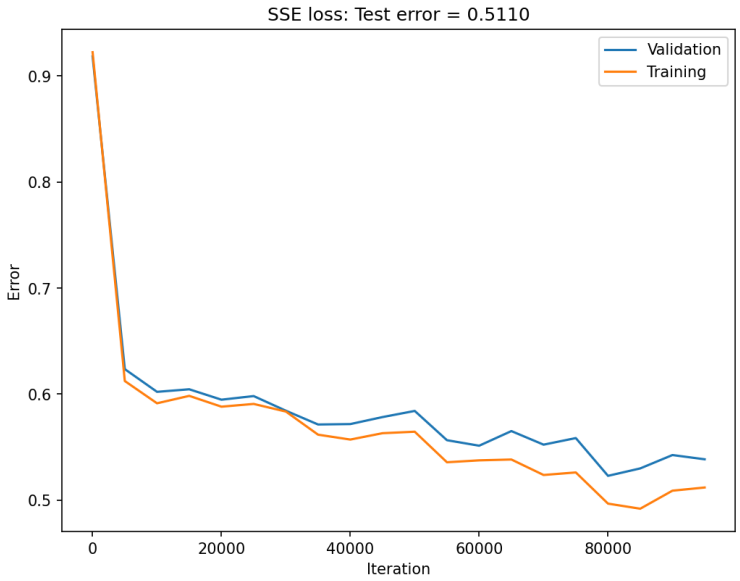

图 1: Learning Curve (Baseline)

Q3. Vectorize evaluating the loss function.

在按照指导文件对网络进行构建时，我发现样例代码的结构存在较多的问题，例如没有充分利用矩阵运算，循环结构很多，不同函数之间有代码重叠和冗余，导致时间复杂度很高。这会极大地影响模型训练的效率，因此在进行后续任务前首先对样例代码进行优化是非常有必要的。下面将按照代码运行顺序逐一修改。

1. Data Preprocessing: 在数据预处理这一部分，样例代码直接将 bias 与输入数据合并，使网络架构并不清晰直观，且会让对于 bias 的梯度计算变得较为复杂。因此，我删除了合并数据的代码，在后续模型训练中单独定义了 bias，让代码可读性更强。

2. Weight Initialization: 在权重初始化这一部分，样例代码首先计算出所有可训的参数个数，然后随机生成了一个一维向量作为初始权重，并没有包含哪些权重属于哪些层的信息，这会导致后续建模对不同层的权重分别初始化时非常麻烦，延长训练时间。所以我在训练前就定义并初始化好了每层对应的权重，直接传入 MLPclassificationPredict 函数（修改后的）。另外，样例代码将第一层 input layer 中的参数也一起归入 hidden layer 中，但实际上二者的性质并不一致，且在后续优化中会对二者进行区别处理，因此我在这里将 input layer 中的参数单独定义。

3. MLP Predict: 在前向传播这一部分，样例代码每次都将输入的一维权重重构为每一层的权重再进行训练，非常冗余耗时。而在 2 中提前定义好每层权重就能避免重复调用权重运算过程，减少时间开销。为了提高运行效率，将该部分的循环结构修改为更快速的矩阵计算。

$$\text{ip}\{i\} = \text{fp}[i-1]W$$
$$\text{fp}\{i\} = \tanh(\text{ip}[i])$$

4. MLP Loss: 在反向传播这一部分，样例代码的前半部分和 3 完全一致，将该代码段删除改为调用 MLPclassificationPredict 能够让代码更简洁。同样为了提高运行效率，将该部分的循环结构修改为更快速的矩阵计算，且利用了矩阵的迹 (trace) 的性质来简化矩阵求导，即 $\text{tr}(A^TB) = \text{tr}(B^TA), \text{tr}(AB) = \text{tr}(BA)$。下面是简单的推导。

   当损失函数为平方损失 SSE，即 $\text{Loss} = (y-\hat{y})^T(y-\hat{y})$ 时，由链式法则，有：

$$\begin{aligned}\Delta\text{Loss} &= \text{tr}\left(\Delta\hat{y}^T\nabla\,\text{tr}\left[(y-\hat{y})^T(y-\hat{y})\right]\right)\\ &= 2\,\text{tr}\left((\Delta\hat{y})^T(y-\hat{y})\right)\\ &= 2\,\text{tr}\left((\Delta(\text{fp}[\text{end}]W))^T(y-\hat{y})\right)\\ &= 2\,\text{tr}\left(\Delta W^T\,\text{fp}[\text{end}]^T(y-\hat{y})\right)\end{aligned}$$

   则 Loss 关于输出层权重 W 的导数为 $2\,\text{fp}[\text{end}]^T(y-\hat{y})$，继续使用链式法则，又有：

$$\begin{aligned}\Delta\text{Loss} &= 2\,\text{tr}\left(W^T\Delta\,\text{fp}[\text{end}]^T(y-\hat{y})\right)\\ &= 2\,\text{tr}\left(\Delta\,\text{fp}[\text{end}]^T(y-\hat{y})W^T\right)\\ &= 2\,\text{tr}\left(\left(\Delta\,\text{ip}[\text{end}]\odot\text{sech}^2(\text{ip}[\text{end}])^T(y-\hat{y})W^T\right)\right.\\ &= 2\,\text{tr}\left(\Delta\,\text{ip}[\text{end}]^T\{\text{sech}^2(\text{ip}[\text{end}])\odot((y-\hat{y})W^T)\}\right)\end{aligned}$$

   故 Loss 关于 ip[end] 的导数为 $2\,\text{sech}^2(\text{ip}[\text{end}])\odot((y-\hat{y})W^T)$。在此基础上不断应用链式法则即可计算得到所有的梯度。

在模型向量化之后，MLPclassificationLoss 函数的代码变得非常精炼，多处调用 numpy 类科学计算方法，除了反向传播计算梯度的部分，其他都避免使用循环结构，算法效率得到了极大的提升。

[ ]:
```
# Output Weights
gOutput = np.dot(fp[-1].T, err)
err = np.multiply(sech_square(ip[-1]), np.dot(err, outputWeights.T))
# Hidden Weights
gHidden = [0] * len(hiddenWeights)
for h in range(len(hiddenWeights) - 1, -1, -1):
    gHidden[h] = np.dot(fp[h].T, err)
    err = np.multiply(sech_square(ip[h]), np.dot(err, hiddenWeights[h].T))
# Input Weights
```

```
gInput = np.dot(np.concatenate((np.ones((nInstances, 1)), X), axis = 1).T, err)
g = [gInput, gHidden, gOutput]
```

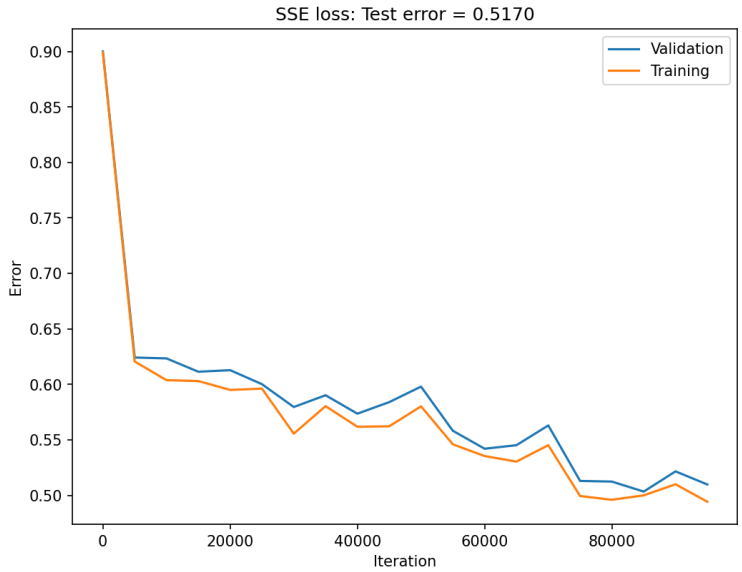

图 2: Learning Curve (Vectorization)

用样例代码和向量化后的代码分别构建包含 10 个节点的单层神经网络，且最大迭代次数均为 100000，发现算法效率得到了显著提高，速度变为原来的 4 倍。这说明网络架构对于训练的重要性，以及优化结构后算法的有效性。

[ ]:
```
%time %run Q0_Baseline\example_quiet.py
```

```
CPU times: total: 12 s
Wall time: 12 s
```

[ ]:
```
%time %run Q3_Vectorization\example_quiet.py
```

```
CPU times: total: 3.03 s
Wall time: 2.8 s
```

Q1. Change the number of hidden units.

将单层神经网络封装为一个函数，参数 inputFeatures 表示输入层中的单元数（输入层也是隐藏层）。结果显示模型预测效果明显随着输入层中单元数变大而变好，时间也更长，这是由于可训参数增加，需要计算的梯度和权重数量成倍增长；而收敛速度则随着单元数的增加而变慢，这是由随机梯度下降本身的性质带来的。但是当节点数大于 200 后，错误率没有特别显著的下降，时间复杂度却变得很高。因此，我考虑在后续模型改造中先使用时间复杂度不高、预测效果相对较好的 200 作为默认

的节点个数，在模型搭建得比较完善之后再进行最后的调参。这也是为什么我这里只初步查看了单层神经网络随节点数变化的预测效果，多层神经网络将在最终模型调参中被尝试或应用。

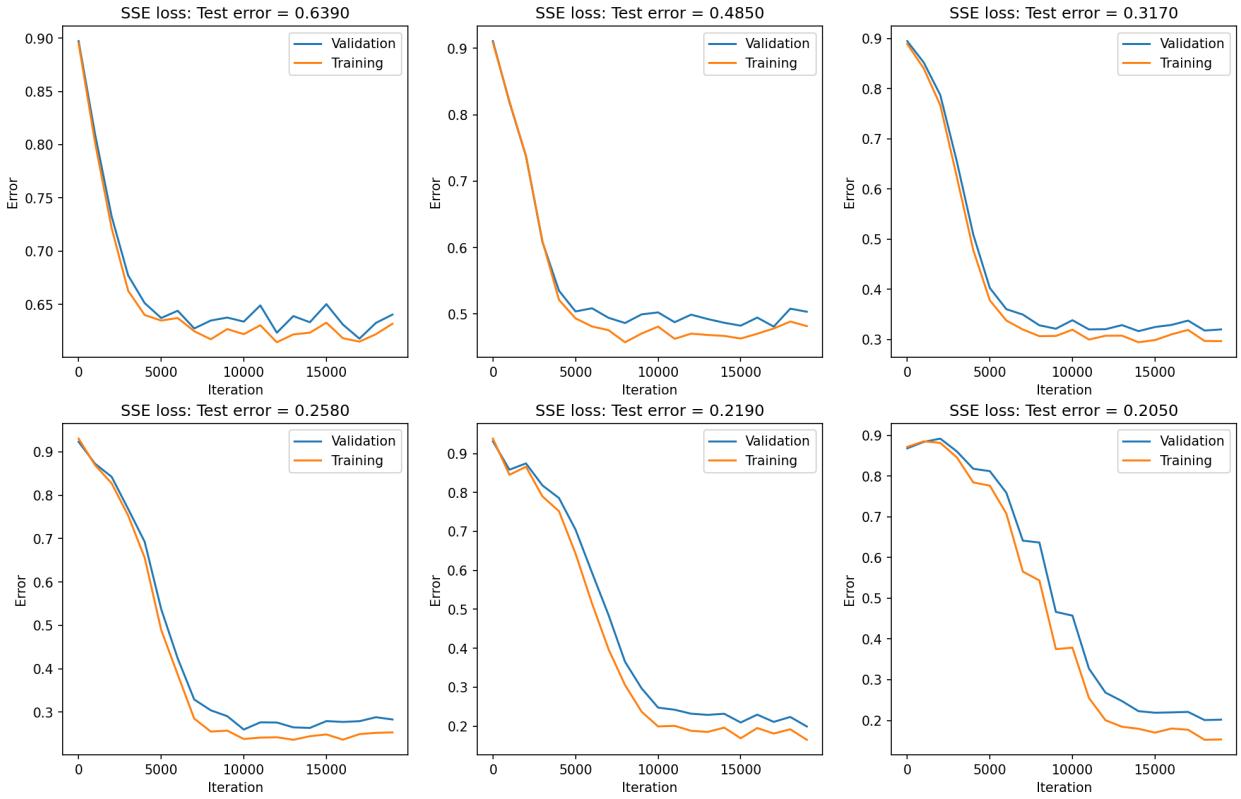

图 3: Learning Curve of Different Networks

Q6. Introduce bias to each layer.

bias 的主要作用是当某一层 layer 的输入均值偏离 0 时，给予模型偏置补偿，加快神经网络的收敛。加入 bias 会带来少量新的参数，能在不牺牲时间复杂度的情况下提高拟合效果。之前的网络只在最开始的 input 层加入了 bias，而隐藏层、输出层都没有添加 bias，这可能会导致激活函数失效。这个项目使用的是输入输出符号相同的非线性映射 tanh，如果输入的分布中心偏离 0 太远，就会导致输出的符号基本相同，让模型收敛速度变慢。

[ ]:
```
bias = np.ones((nInstances, 1))
X -> np.concatenate((bias, X), axis = 1)
```

其他结构保持不变，在每一层都加入 bias，发现神经网络的训练效果并没有特别明显的提升（多次实验取平均，大概只提升 2% 左右）。但因为 bias 并不会给神经网络增加太大的计算压力，在效果上也有少许提升，我认为给每层加入 bias 是合适的。

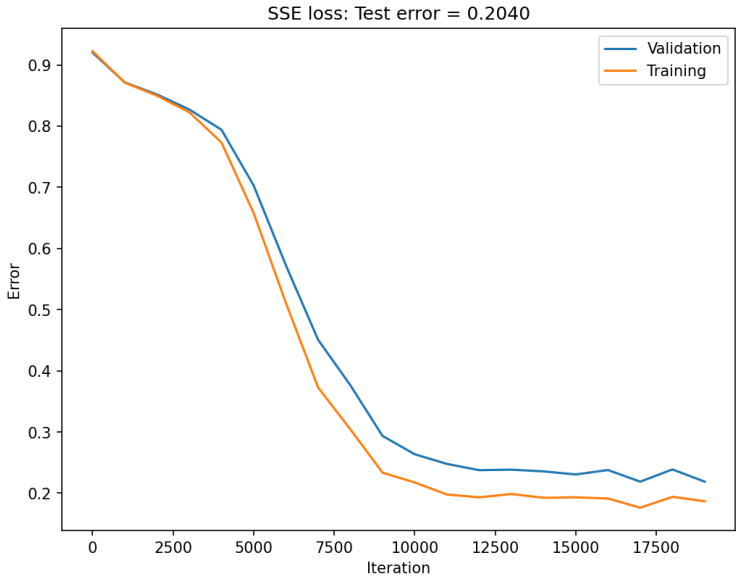

图 4: Learning Curve (Bias)

Q2. Momentum

随机梯度下降的收敛速度有时会比较慢，因为当接近最优值时梯度会比较小，而学习率 α 固定，就可能会陷入局部最优。动量算法的提出就是为了加速模型学习，特别是处理高曲率、小但一致的梯度，或是带噪声的梯度。动量算法积累了之前梯度指数级衰减的移动平均，并且继续沿该方向移动，引导参数向最优值更快收敛，更新公式如下：

$$w^{t+1} = w^t - \alpha_t \nabla f(w^t) + \beta_t (w^t - w^{t-1})$$

结果显示加入动量和不加入动量的错误率基本一致，但前者收敛速度明显快于后者，说明动量算法能够较好地改进收敛率。

[ ]:
```
def momentum(g, Weights, prev_Weights, alpha, beta):
    """Update weights with momentum."""
    if it != 0:
        Weights[2] -= alpha * g[2] - beta * (Weights[2] - prev_Weights[2])
        Weights[0] -= alpha * g[0] - beta * (Weights[0] - prev_Weights[0])
        if len(Weights[1]) != 0:
            Weights[1] = [Weights[1][h] - alpha * g[1][h] + beta *␣
↪(Weights[1][h] - prev_Weights[1][h]) for h in range(len(Weights[1]))]
    else:
        Weights[2] -= alpha * g[2]
        Weights[0] -= alpha * g[0]
```

```
        if len(Weights[1]) != 0:
            Weights[1] = [Weights[1][h] - alpha * g[1][h] for h in␣
↪range(len(Weights[1]))]
    return Weights
```

图 5: Learning Curve (Momentum)

Extra1. Adam

出于让模型加速收敛的目的，我试图调整过 SSE 损失 SGD 的学习率，发现学习率小于 1e-3 时模型收敛速度很慢，学习率大于 1e-3 时又难以收敛，发生剧烈波动或陷入局部最优，这说明在学习率的选择上目前的模型并不稳健。我在这里引入一种学习率自适应的优化算法 Adam(Adaptive moments)，Adam 将动量直接并入了梯度一阶矩（指数加权）的估计，并对从原点初始化的一阶矩（动量项）和（非中心的）二阶矩的估计进行了修正，减小参数的波动幅度。

Adam 的算法步骤为：

1. 计算梯度：g
2. 更新有偏一阶矩估计：$S_t = \beta_1 S_{t-1} + (1-\beta_1)g$
3. 更新有偏二阶矩估计：$R_t = \beta_2 R_{t-1} + (1-\beta_2)g \odot g$
4. 修正一阶矩的偏差：$\hat{S}_t = S_t/(1-\beta_1^t)$
5. 修正二阶矩的偏差：$\hat{R}_t = R_t/(1-\beta_2^t)$

6. 计算更新：$\theta_t = \rho * \theta_{t-1} - \alpha \frac{\hat{S}_t}{\sqrt{\hat{R}_t + \delta}}$（逐元素应用操作）

7. 更新权重：$W_t = W_{t-1} + \theta_t$

我为 Adam 的权重更新定义了一个新的函数 adam，过程中我遇到了两个需要注意的地方。

1. 第 3 步受到数据随机性的影响，更新后的 R_t 可能会非常小，导致在第 6 步计算更新时 θ_t 发生爆炸。因此我加入了一步判断条件，只有当新 R_t 的 2 范数大于旧 R_t 的 2 范数时，才更新 R_t，否则保持原来的值，以避免运算中出现问题。

2. 上述第 4、5 步的偏置修正因子必须在第 6 步中加入，不能分步操作，否则由于计算机计算精度的问题，矩阵会在运算过程中爆炸，难以正常更新梯度。

[ ]:
```
def adam(g_t, Weights_t, theta_t, S_t, R_t, t, alpha):
    # Internal parameters
    beta1 = 0.9
    beta2 = 0.9999
    rho = 0.9
    # Update by Adam
    S_t = beta1 * S_t + (1 - beta1) * g_t
    R_prev = np.array(R_t, copy = True)
    R_t = (beta2 * R_t + (1 - beta2) * (g_t ** 2))
    if (R_t ** 2).sum() == np.nan or (R_t ** 2).sum() < (R_prev ** 2).sum():
        R_t = R_prev
    theta_t = rho * theta_t - alpha / (1 - beta1 ** t) * S_t / (np.sqrt(R_t /␣
↪(1 - beta2 ** t) + 1e-7))
    Weights_t = Weights_t + theta_t
    return Weights_t, theta_t, S_t, R_t
```

结果显示 Adam 算法的收敛速度确实比正常的动量法要快，但也可以观察到该算法后期的收敛效果并不好，尾部一直在反复震荡，难以进入全局最优解。这可能是因为学习率对于后期训练依然过大，可以考虑加入学习率衰减的策略；也有可能是受到二阶动量的影响，每次随机到的数据差异很大时，R_t 就会时大时小而非单调变化，引起学习率的震荡，从而导致模型无法收敛。

Extra2. Xavier Initialization

上述问题可以通过对权重进行 Xavier 初始化来解决。优秀的初始化应该使得各层激活值、状态梯度的方差与传播过程中的方差保持一致，即要保证前向传播各层参数的方差和反向传播时各层参数的方差一致，这样能提高数据的稳定性并改善收敛效率。该初始化方法其实很简单，只需要在每一

图 6: Learning Curve (Adam)

层的初始权重乘上 $\sqrt{\frac{2}{\text{该层节点数} + \text{前一层节点数}}}$ 即可。事实上该方法非常有效，收敛速度非常快，且测试集上的错误率减少到了 4.30%，是目前所有模型中效果最好的。于此同时，训练集和验证集上的错误率差距也明显减小，说明该网络已经在逐步接近数据的真正模型。

[ ]:
```
# Input Weights
prev_layer = d + 1
inputWeights = np.random.randn(prev_layer, inputFeatures) * np.sqrt(2 /␣
 ↪(prev_layer + inputFeatures))
# Hidden Weights
prev_layer = inputFeatures + 1
hiddenWeights = []
for h in range(len(nHidden)):
    hiddenWeights.append(np.random.randn(prev_layer, nHidden[h]) * np.sqrt(2 /␣
 ↪(prev_layer + nHidden[h])))
    prev_layer = nHidden[h] + 1
# Output Weights
outputWeights = np.random.randn(prev_layer, nLabels) * np.sqrt(2 / (prev_layer␣
 ↪+ nLabels))
Weights = [inputWeights, hiddenWeights, outputWeights]
```

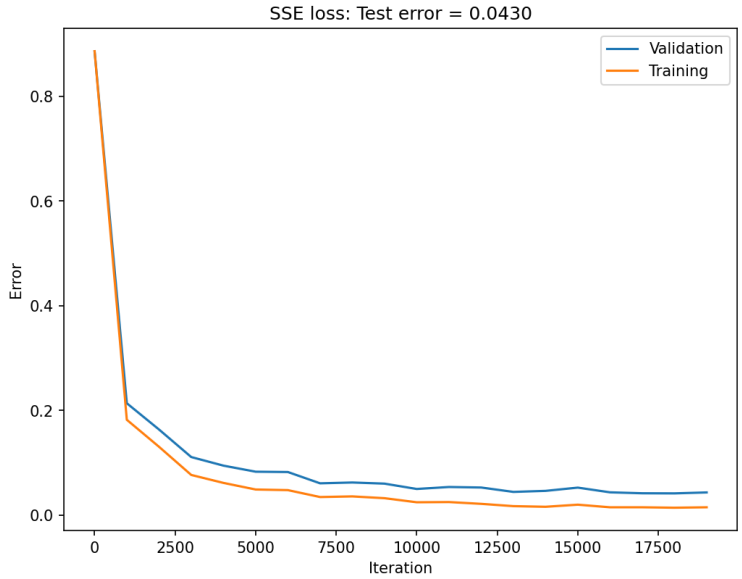

图 7: Learning Curve (Xavier Initialization)

Q5. Softmax Layer and Cross-Entropy Loss.

数据稳定性和收敛率的问题同样可以通过增加 softmax 层来解决。一般情况下，在多分类问题中应用 softmax 函数能得到更好的效果，因为不论网络最终输出的取值范围是多少，softmax 函数都可以将其归一化到 [0, 1]，转化为一个概率分布。添加 softmax 层只需要在神经网络原本的输出层后做一次 softmax 操作，将新的结果作为分类的概率预测值。计算时需要注意 softmax 的计算稳定性问题，当指数过大，softmax 计算可能会爆炸，因此一个合理的操作是在 softmax 分子分母的指数减去同一个常数，在结果不变的情况下让算法更加稳定，即：

$$
\begin{aligned}
p(y_i) &= \frac{\exp(z_i)}{\sum_{j=1}^{J} \exp(z_j)} \\
&= \frac{\exp(z_i - \max\{z_1, ..., z_J\})}{\sum_{j=1}^{J} \exp(z_j - \max\{z_1, ..., z_J\})}
\end{aligned}
$$

在 softmax 场景下，SSE 的标签 {-1,1} 并不适用，需要将其改为 {0,1} 标签，因此我修改了 linearInd2Binary 函数。

[ ]:
```
def linearInd2Binary(ind, nLabels):
    """Transform the labels into {0, 1} encoding."""
    n = len(ind)
    y = np.zeros((n, nLabels))
    for i in range(0, n):
        y[i, ind[i, 0] - 1] = 1
    return y
```

指导文件中提示的损失函数为某类别的负对数似然，也就是预测概率的负对数 $-\log p(y_i) = -\log(\hat{y}_i)$。其反向传播的计算公式为：

$$\begin{aligned}
\nabla_{f_j}\text{Loss} &= \nabla_{f_j}(-\log \hat{y}_i) \\
&= -\nabla_{f_j}\log\left(\frac{\exp(f_i)}{\sum_j \exp(f_j)}\right) \\
&= -\nabla_{f_j}\left(f_i - \log\sum_j \exp(f_j)\right) \\
&= \begin{cases} \frac{\exp(f_i)}{\sum_j \exp(f_j)} - 1 & j = i \\ \frac{\exp(f_j)}{\sum_j \exp(f_j)} & j \neq i \end{cases} \\
&= \hat{y}_j - y_j
\end{aligned}$$

可见反向传播中损失函数关于输出层的导数为 $\hat{y} - y$，和 SSE 的导数相比只是符号相反、少了系数 2，这让代码调整变得十分简单。该损失和 softmax 场景下常用的交叉熵损失函数是等同的，推导如下：

$$H = -\sum_{j=1}^{J} y_j \log(\hat{y}_j) = -\log(\hat{y}_i) - \sum_{j\neq i} 0 * \log(\hat{y}_j) = -\log(\hat{y}_i)$$

因此为了提高运算速度，我在这里使用交叉熵作为损失函数，利用矩阵运算快速求解。

[ ]:
```
def softmax(yhat, y):
    yhat = yhat.T - np.max(yhat.T, axis = 0)
    s = np.exp(yhat)
    s = (s / np.sum(s, axis = 0)).T
    s = np.multiply(- np.log(s), y).sum()
    return s
```

结果显示，加入 softmax 层后，网络的收敛速度又加快了，且训练曲线的波动性减少，变得更加平滑。SSE 损失的网络大约在 4000 次迭代后错误率才跌破 10%，而这里的网络在 2000 次迭代后便超越了前者，最终的错误率仅为 3.9%，比起上一个网络又有提高，还是在没有调整任何参数的情况下。这得益于 softmax 将输出转化为概率，且引入了指数函数以扩大分布间的差异性，不但能使模型有更好的鲁棒性，也能增加模型精度。

Extra3. Minibatch

为了进一步加快收敛速度并提高稳定性，我增加了随机梯度下降每次求梯度的样本量。原先只是对随机到的单样本求梯度，这里我选择了 20 作为 minibatch 的大小。我们知道随机梯度下降的核心在于梯度是期望，如果只选择一个样本进行梯度估计，是很难对真正的梯度进行近似的，因此训练的方差就会很大，算法不太稳定。增加 minibatch 之后，可以在算法的每一步从训练集中均匀抽出

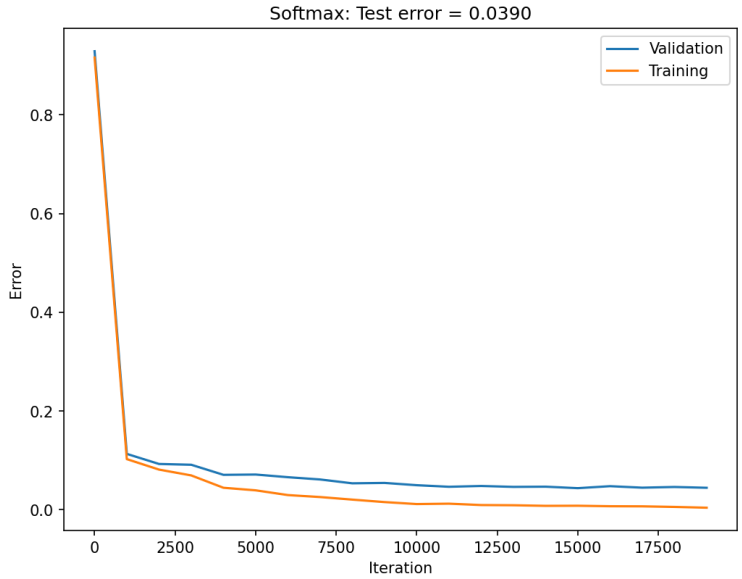

图 8: Learning Curve (Softmax)

一小批量的样本，这样梯度的变化就会比较平稳，而且训练模型用到的样本量也增加了，收敛速度随之提高。由于之前在 Q3 中已经对代码进行向量化处理，因此只需要修改 minibatch 的参数大小以及均匀生成样本索引的随机函数即可。

结果显示，收敛速度大幅提高，原先需要 10000 次迭代左右才能达到的错误率，在加入 minibatch 之后至多需要 1000 次就能达到，且后续的训练曲线非常平稳，几乎是一条直线，最终测试集上的错误率也有所下降，效果比较好。

不过这里的模型仍然有一些问题。小批量的梯度下降不可避免地会带来大量的矩阵运算，从而增加时间复杂度。另外，随着 minibatch 大小的增加，过拟合的风险也会增加，从图像中可以看出，模型在训练集与验证集上错误率依然存在不小的差距。因此，在下一个任务中完成 Fine Tuning 之后，会有几个任务专注于增加模型的泛化性，如添加 L2 正则化项、early stopping、dropout、数据增强等。

[ ]:
```
i = np.ceil(np.random.uniform(0, n, minibatch)).astype(int)
```

Q8. Fine Tuning

每一次迭代时，保持其他层的参数不变，对输出层的权重进行微调 (fine tuning)。这相当于在整体网络优化中间加了一步小的优化，能够不牺牲太多计算复杂度就达到提高模型准确率的效果。微调的主要步骤为：

1. 保持 fp[end] 不变，在此基础上求解最小化损失函数的输出层权重 W。

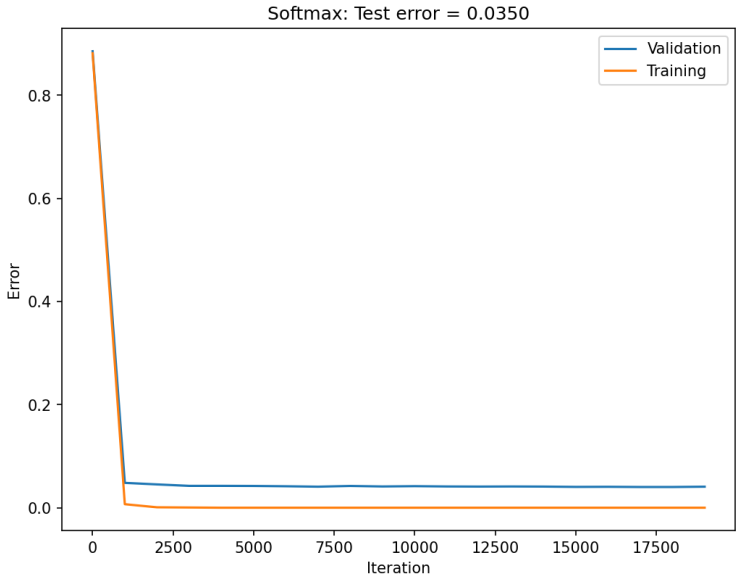

图 9: Learning Curve (Minibatch)

2. 使用更新后权重计算得到的 $\hat{y}$ 计算误差项 $err = \hat{y} - y$，然后按正常流程进行反向传播，得到所有的梯度。

3. 达到指定的最大迭代次数后，使用全部的训练数据对输出层权重进行微调，得到最终模型。

在定义微调代码时有几个需要注意的地方：

1. 受到随机性影响，如果在模型最初训练时就加入 fine tuning 步骤，那么可能达不到增加收敛速率的目的。即使让当前输出层很好地逼近当前的 minibatch，也并不代表就能在验证集上表现得好，反而会出现非常严重的过拟合问题。因此，我设置了一个阈值 2000，当模型迭代次数大于该阈值后，再加入 fine tuning。此时的模型权重已经基本收敛，保持稳定，微调不再会导致权重的大幅度变动，节省了微调步的运算时间，且有利于帮助模型跳出局部最优。

2. 在微调中的优化部分，我依然选择了 Adam 方法更新权重，它的效果仅依赖于一个合适的学习率。实际上还有许多可以使用的优化方法，如最小二乘法、拟牛顿法、BFGS 等，但是应用于引入 Softmax 层的网络，稳健性、收敛率没有 Adam 方法高。最小二乘法主要适用于 SSE 损失的网络，而其他需要求解二阶 Hessian 矩阵的优化方法计算压力较大，且一旦某次迭代出现了奇异矩阵，整个网络就训练失败了。Adam 方法能够避免在优化时出现这些问题。

3. 在计算损失函数时，由于对数函数的存在和计算机精度的问题，需要对 $\hat{y}$ 加一个微小常量避免程序报错。

结果显示，微调比不微调的准确率更高一点，但也仅仅是一点，训练曲线同样平稳，基本没有大的差别。这可能是因为模型出现了过拟合的问题，比如训练数据拟合得太好，训练集准确率提高的同

时模型泛化能力降低，测试集上的准确率相对较低。下面我就来解决这个问题。

[ ]:
```
def fine_tune(yhat, y):
    return np.multiply(- np.log(yhat + 1e-6), y).sum()
```

[ ]:
```
# Output Weights
if it > 2000:
    while (fine_tune(yhat, y) - f_prev) ** 2 >= 1e-5:
        f_prev = fine_tune(yhat, y)
        g = np.dot(np.concatenate((bias, fp[-1]), axis = 1).T, yhat - y)
        outputWeights, theta, S, R = adam(g, outputWeights, theta, S, R, t,␣
 ↪1e-5)
        yhat = np.dot(np.concatenate((bias, fp[-1]), axis = 1), outputWeights)
        yhat = yhat.T - np.max(yhat.T, axis = 0)
        s = np.exp(yhat)
        yhat = (s / np.sum(s, axis = 0)).T
        t += 1
    gOutput = outputWeights - temp
else:
    gOutput = np.dot(np.concatenate((bias, fp[-1]), axis = 1).T, yhat - y)
```

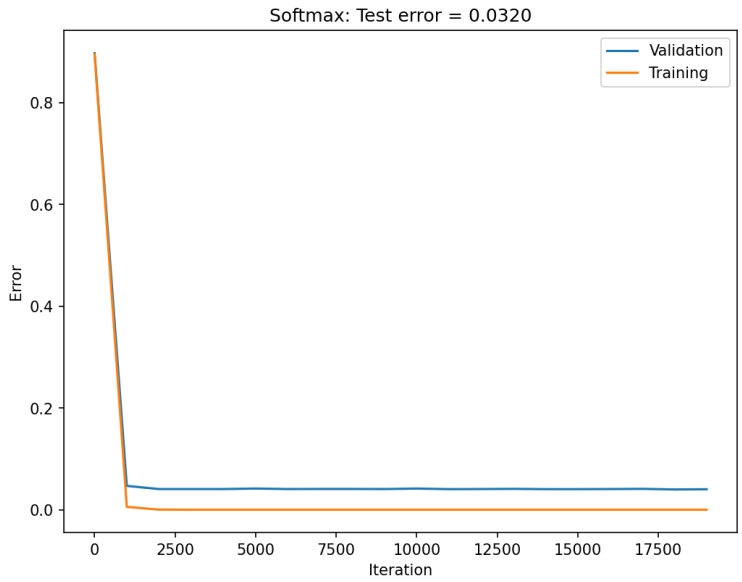

图 10: Learning Curve (Fine Tuning)

Q4. Regularization: L2 and Early Stopping

这里加入的是关于权重的 L2 正则项，只需要在反向传播代码（含 Fine Tuning 部分）中添加 λW 即可。结果显示加入正则化的网络和没有惩罚的网络基本一致，没有明显增加模型的泛化能力。这可能和 λ 参数的取值有关，不过我暂时不展开讨论，等模型基本构建完毕后再进行调试。

[ ]:
```
def fine_tune(yhat, y, w):
    return np.multiply(- np.log(yhat + 1e-6), y).sum() + penalty / 2 * (np.
 ↪linalg.norm(w, 2) ** 2)
```

[ ]:
```
gOutput = gOutput + penalty * outputWeights
for h in range(len(hiddenWeights) - 1, -1, -1):
    gHidden[h] = gHidden[h] + penalty * hiddenWeights[h]
gInput = gInput + penalty * inputWeights
```

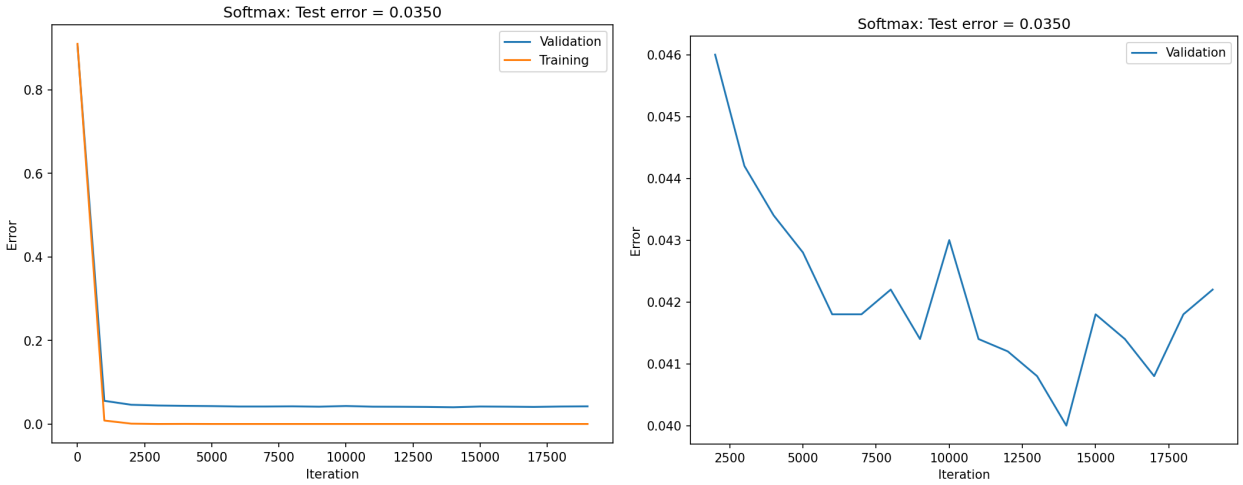

图 11: Learning Curve (L2 Regularization)

在上图中，我们发现随着迭代次数的增加，训练误差始终保持在 0 附近，但验证集的误差再次上升。这是因为该模型有很强的拟合能力，造成了过拟合现象的发生。早停法就是针对这种情况的一种简单而有效的正则化方式，核心思想是返回使验证集误差最低的参数设置，以此有希望获得更好的测试误差。它几乎不需要改变基本训练过程、目标函数或一组允许的参数值，实现方法非常简单，就是在每次验证集误差有所改善后，用 Wbest 存储模型参数的副本，当训练算法终止时，返回 Wbest 中的参数而不是最新的参数。

我在这个方法的基础上又做了一些小修改，让每次训练不使用前一轮的权重，直接将 Wbest 作为当前的权重。为了增加模型的稳健性，我将 Wbest 的更新条件从验证集误差低于最好的误差时保存参数修改为验证集误差低于最好误差 + 一个小常数（这里是 5e-4）时就保存参数，这综合避免了模型过拟合和陷入局部最优两个问题，使验证集在有一定波动性的情况下逐渐收敛到最优。该算

法还可以借鉴模拟退火算法的思路，当验证集误差小于最优时以 1 的概率更新最优权重，当验证集误差大于最优时以一定概率（$\exp -\Delta(\epsilon/\tau)$，其中 τ 随迭代次数变大而减小）更新最优权重，有利于快速收敛和跳出局部最优。这两种最优权重更新方法我都尝试了，发现结果基本一致，而模拟退火还需要调超参数比较麻烦，因此还是使用绝对误差判断。

[ ]:
```
best = 1
if np.sum(yhat != (yvalid - 1)[:, 0]) / t - best <= 5e-4:
    Wbest = Weights[:]
    best = np.sum(yhat != (yvalid - 1)[:, 0]) / t
else:
    Weights = Wbest[:]
```

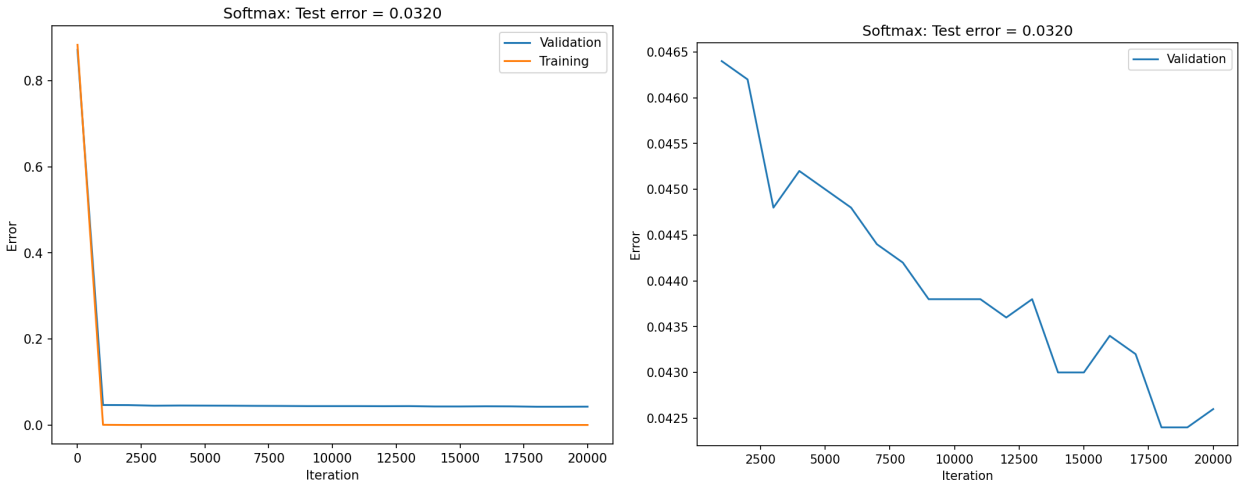

图 12: Learning Curve (Early Stopping)

Q7. Dropout

暂退法也是一个常用的正则化方法，即在每次训练中随机地丢弃一些神经元，而在测试中则不丢弃任何节点，因为部分神经元可能是冗余的，丢弃之后会让另一部分的神经元拟合得更好。这在一篇论文中被类比为有性繁殖，提出了 “共适应性” 的概念，即神经网络过拟合与每一层都依赖于前一层的激活值相关，而暂退法会破坏共适应性，从而减少结构化风险。需要注意的是，在训练过程中 dropout 会带来偏差，需要对保留节点的分数进行规范化来消除每一层的偏差，即让保留节点除以（1-暂退概率 p）。

暂退法的反向传播比较容易修改，推导如下：

$$\begin{aligned}
\Delta\text{Loss} &= \operatorname{tr}\left((\Delta(\text{fp[end]}W))^T(y-\hat{y})\right) \\
&= \operatorname{tr}\left(\Delta W^T\,\text{fp[end]}^T(y-\hat{y})\right) \\
&= \operatorname{tr}\left(\Delta\,\text{fp[end]}^T(y-\hat{y})W^T\right) \\
&= \operatorname{tr}\left(\left(\Delta\,\text{ip[end]}\odot\text{Mask}\odot\operatorname{sech}^2(\text{ip[end]})\right)^T(y-\hat{y})W^T\right) \\
&= \operatorname{tr}\left(\Delta\,\text{ip[end]}^T\{\text{Mask}\odot\operatorname{sech}^2(\text{ip[end]})\odot((y-\hat{y})W^T)\}\right)
\end{aligned}$$

故 Loss 关于 ip[end] 的导数为 $\text{Mask}\odot\operatorname{sech}^2(\text{ip[end]})\odot((y-\hat{y})W^T)$。

结果显示，加入 dropout 后的模型效果变差，虽然稍微减小了训练集和验证集误差的差距，但是这牺牲了对训练集的拟合程度，反而让整体模型的错误率上升了。此外，加入 dropout 也让验证集错误率的变动变得非常不平稳，不利于模型的训练，因此在后续建模中将不使用 dropout 方法。

[ ]:
```python
def mask(X, p, train = True):
    _, nVars = X.shape
    if train:
        return (np.random.randn(nVars) > p) * X / (1 - p)
    else:
        return X
```

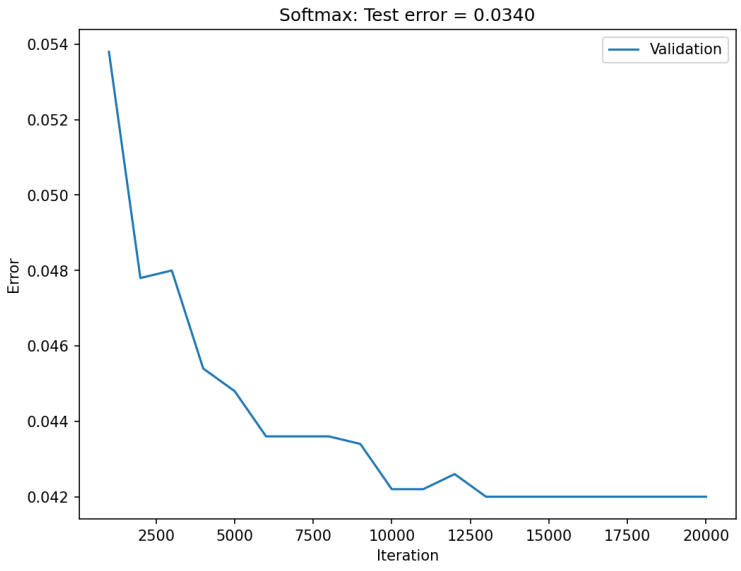

图 13: Learning Curve (Dropout)

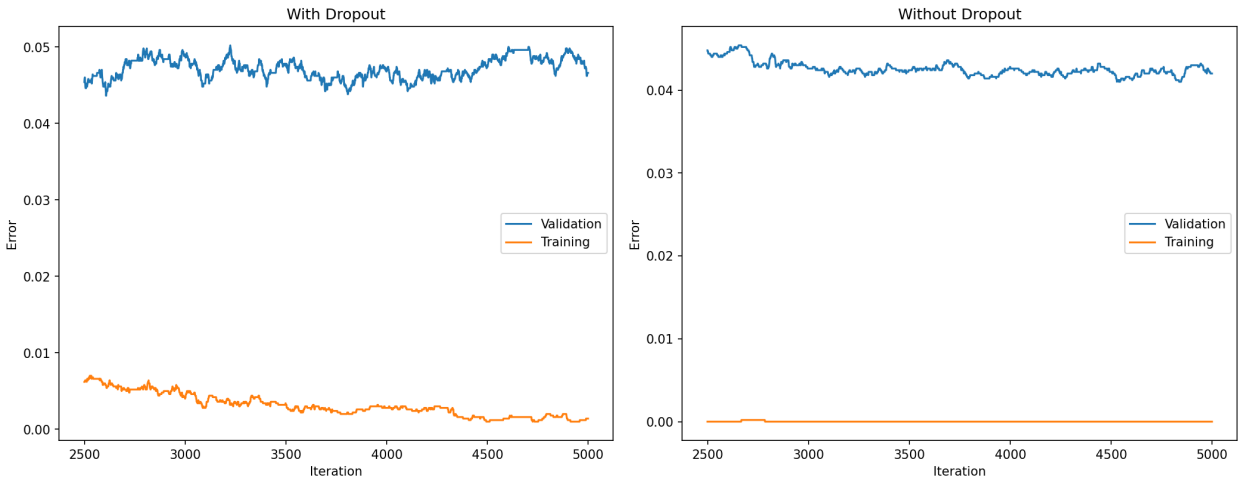

图 14: Learning Curve (Dropout)

Q9. Data Augmentation

让机器学习模型泛化得更好的最好办法是使用更多的数据进行训练，数据增强就是解决这个问题的一种方法，让我们可以利用有限数据创建假数据并添加到训练集中进行训练。需要注意的是，我们不能使用会改变类别的转换，例如随机旋转角度不能太大，否则会混淆 “6” 与 “9” 之间的区别；也不能裁剪太多，否则会减少类别的特征。我在这里设置了两种数据数据增强的方法，它们全部基于 $[-20, 20]$ 度的旋转，之后通过参数 crop 指定裁剪或填补（整张图片的中位数），使用 Image 库对填补后大小改变的图像 resize 为原来大小 16*16，效果如下图所示。

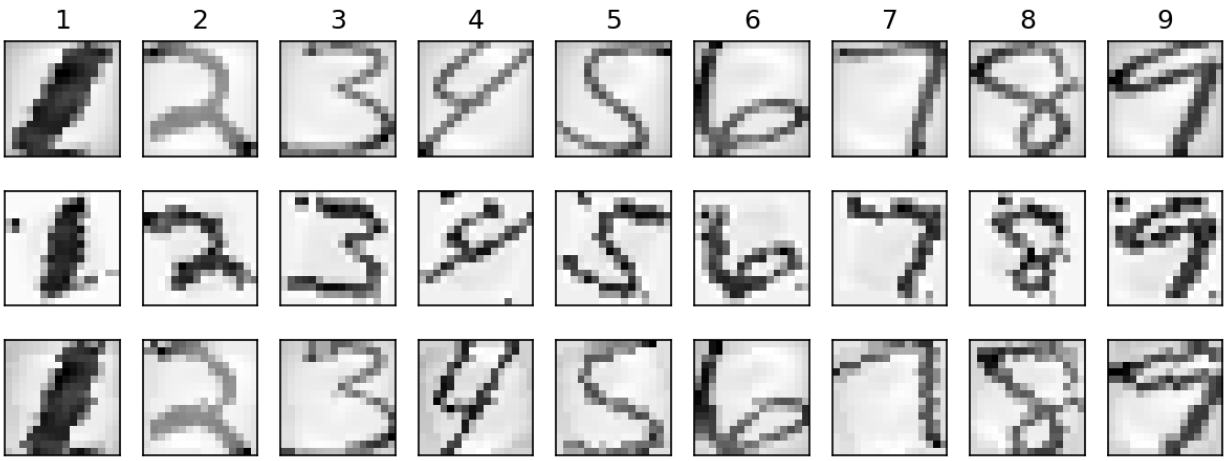

图 15: Augmented Data

将训练集数据增强 20 倍，数据量为 100000，其中 5000 为初始训练集，50000 个样本来自于非填补式数据增强，另 45000 个样本来自于填补式数据增强。将随机生成的数据以 matlab 格式保存至本地，命名为’newdigits.mat’。读取新的数据，使用不带 dropout 的方法，发现收敛速度虽有降低，

但拟合效果得到了很大的提高，错误率降低了约 1%，且过拟合问题也得到了很好的解决，说明数据增强非常有效地保留了类别的重要特征。在这里我只对数据进行了 final tuning，没有在每次训练时增加 fine tuning，因为数据增强后针对 minibatch 微调很容易造成过拟合，与数据增强的目的相悖，此外这还会带来很大的时间开销，不是很合适。

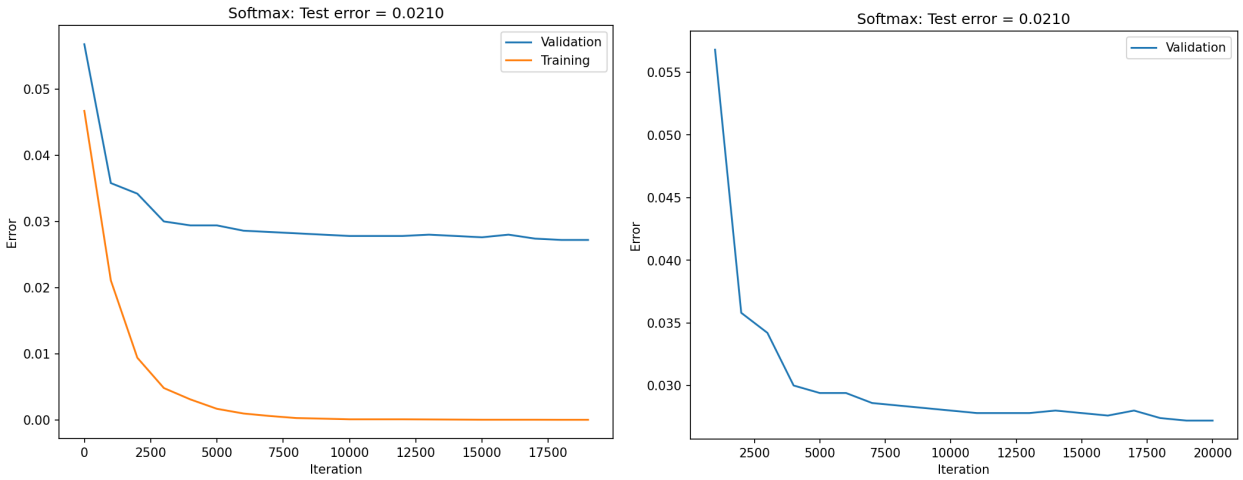

图 16: Learning Curve (Data Augmentation)

Extra4. RELU

激活函数分为饱和激活函数和非饱和激活函数，上面任务中使用的 tanh 函数就是饱和激活函数，只对接近 0 的输入敏感，而这种饱和性会使基于梯度的学习非常困难，容易导致梯度消失问题。例如当输入为绝对值很大的值时，输出会饱和到一个高值或低值，对应的梯度约等于 0，即使从上一步传导来的梯度较大，该神经元的权重和 bias 的梯度也会趋近于 0，导致参数无法得到有效更新。

使用 RELU 作为激活函数就能有效地解决该问题，其梯度只能为 0 或 1。只要 RELU 处于激活状态，它的一阶导数处处为 1，而二阶导数几乎处处为 0，这意味着相比于引入二阶效应的激活函数来说，它的梯度方向对于学习来说更加有用。此外，它的计算效率很高，只需要做正负的符号判断即可得到梯度；它还有很高的鲁棒性，能够在一定程度上排除噪声干扰。因为输入正值的大小代表检测特征信号的强弱，而负值的大小则表示噪声或者其他特征信息，RELU 能够直接将负值（噪声）截断为 0，避免了无用信息的干扰。

Loss 关于 ip[end] 的导数为 $(\text{ip[end]} > 0) \odot ((y - \hat{y})W^T)$。需要注意使用 RELU 时，权重 Xavier 初始化不再适用，因为 Xavier 假定激活函数在 0 附近为线性函数，而 RELU 并不满足。因此权重初始化时应该使用与 RELU 相适应的 HE 初始化。

因此在这里我尝试将激活函数从 tanh 修改为 RELU，结果有所提高，验证集与训练集的错误率差距缩小到 2% 左右，且验证集与测试集的错误率相差很小。在不调参的情况下，relu 网络相比于 tanh 网络平均提高了 0.3% 的正确率（多次实验取平均）。

```
[ ]: fp[h] = [(ip[h] > 0) * ip[h]]
```

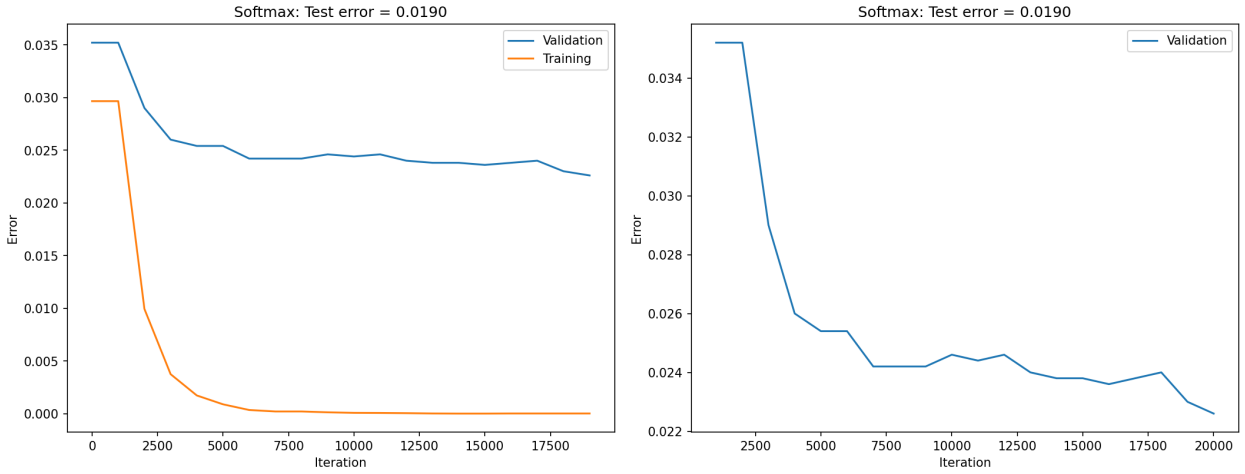

图 17: Learning Curve (RELU)

修改输入层和隐藏层的超参数（节点个数），模型效果又有提升，错误率基本稳定在 1.5% 左右，表现最好的模型错误率为 1.4%，即准确率达到 98.6%。

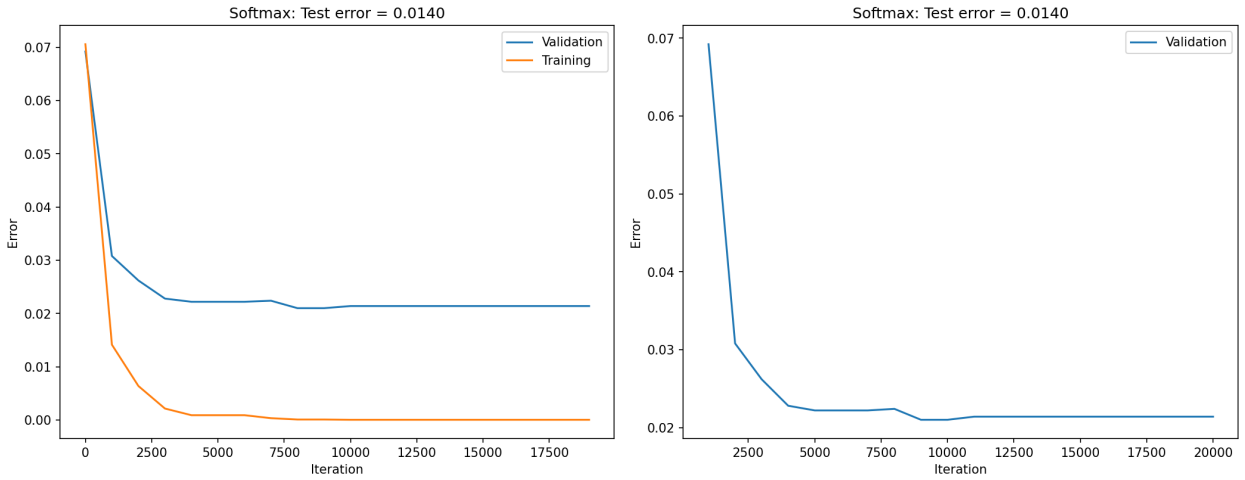

图 18: Learning Curve (RELU)

Q10. Convolution 2D

在原先神经网络的输入与全连接层之间添加一层卷积层，我这里设置的可调节的超参数有两个，分别为卷积核大小（kernelsize）和卷积通道的数量（nChannels），并没有设置 padding 参数和 stride 参数。我没有让卷积之后的数据经过激活函数，直接进入第一层全连接层，因为我在实验中发现，卷积过后数据的分布会发生很大的变化，如果进入激活函数会产生梯度消失的问题，没有办法很好地更新权重。

卷积层的反向传播计算公式为：

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \,\mathrm{ip}[0]} \frac{\partial \,\mathrm{ip}[0]}{\partial W}$$
$$= \mathrm{rot180}(X) * \frac{\partial L}{\partial \,\mathrm{ip}[0]}$$

其中 $\mathrm{rot180}(X)$ 表示将输入矩阵旋转 180 度，$*$ 表示卷积操作。另外，我也没有在这里使用 Adam 优化器，只是在 SGD 的基础上添加了动量，因为经过多次实验 Adam 在这个卷积网络中收敛性较差，其可能原因我在 Extra1 部分已经解释过了，这里不再赘述。

结果显示，CNN 相比于不加卷积层的网络在收敛速度和测试集错误率上并没有提升，反而有所下降，且计算时间开销非常大。一方面是因为卷积操作比较耗时，另一方面可能是卷积后得到的数据特征有很多是冗余的，并不能帮助模型进一步提高。

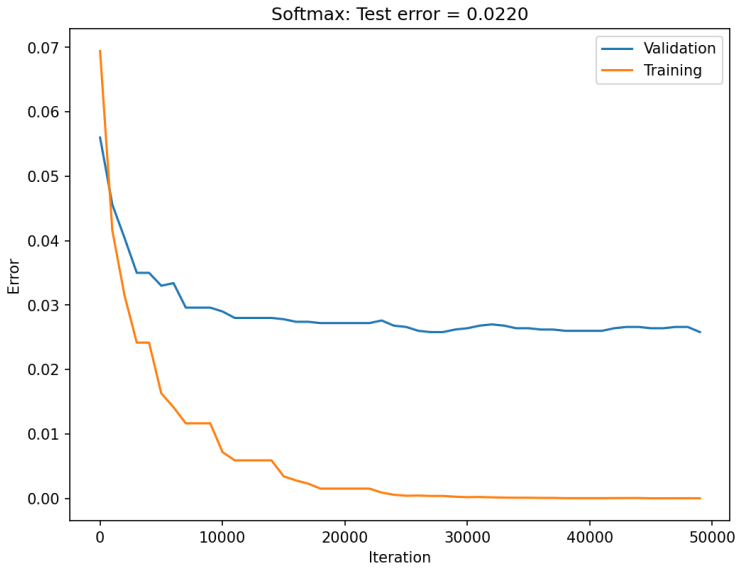

图 19: Learning Curve (Convolution2D)

将模型训练好的权重与模型中的数据进行卷积操作，挑选几张图片查看卷积效果。第一行为原始数据，第二至四行为卷积的 3 个通道，发现提取的特征都比较重复，可能这就是为什么加入卷积层之后没能提高模型效果的原因之一，因此我决定在最终模型中不加入卷积层。

Final. Parameter Tuning

在模型搭建完毕之后，选择表现最好的一个模型进行参数调整。重要参数有 L2 正则化的惩罚系数 λ、是网络结构参数（包含网络层数、每层节点数量），以及 minibatch 参数。

我将程序封装为 FinalModel 函数，固定其他的参数不变，然后对某个参数的不同取值进行测试。

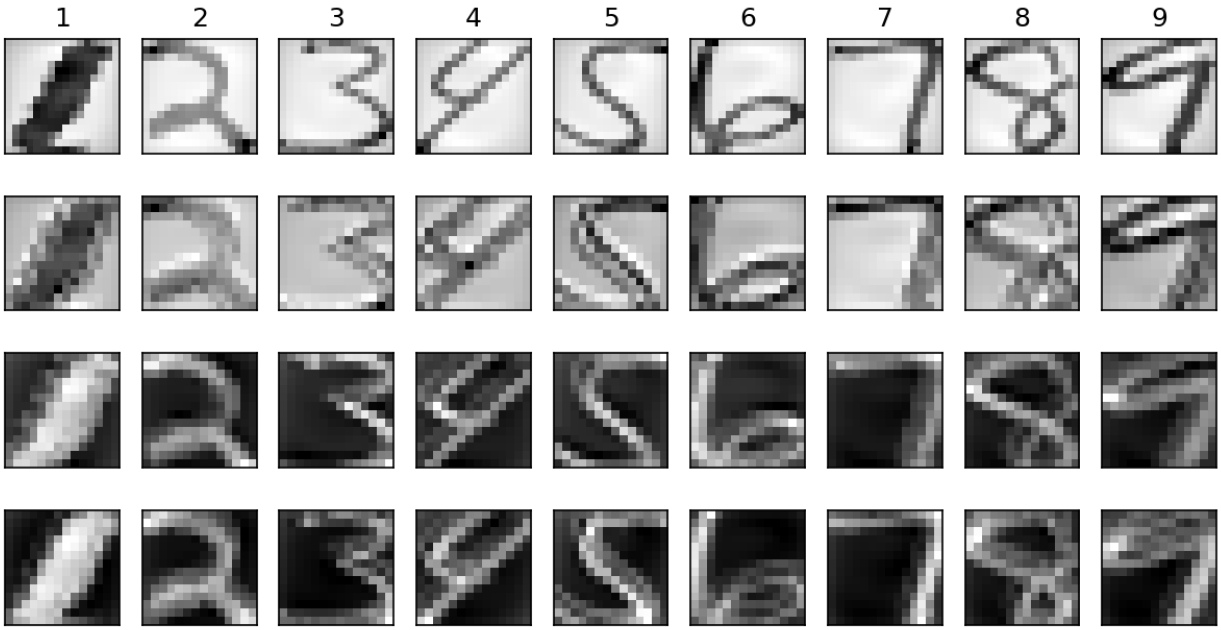

图 20: Convolution 2D Data

1. 单层神经网络：查看节点数量变化对模型结果的影响，发现随着模型节点数的增多，错误率也在不断降低，但当节点数大于 128 之后，错误率并没有很明显的下降。

- 节点数量为 10: 0.068000
- 节点数量为 50: 0.030000
- 节点数量为 128: 0.019000
- 节点数量为 256: 0.018000
- 节点数量为 512: 0.016000

2. 双层神经网络：以效果最好的 256 作为第一层的节点数量（512 的时间开销太大，且没有太明显的提升），查看隐藏层节点数量对模型结果的影响。发现当隐藏层节点为 16 时，多次实验平均下来的错误率最低，因此模型结构固定在双层，第一层有 256 个节点，第二层隐藏层 16 个节点，输出层 10 个节点。

- 节点数量为 16: 0.014000
- 节点数量为 32: 0.017000
- 节点数量为 64: 0.019000
- 节点数量为 128: 0.019000
- 节点数量为 512: 0.016000

3. Minibatch：在目前最优的模型上查看 minibatch 数量对模型结果的影响。结果显示，minibatch 为 100-150 左右时，模型训练效果最优，非常稳定地让错误率保持在 1.5% 左右；当 minibatch

较小时，错误率较高，是因为没能有效地利用数据中的信息；当 minibatch 较大时，错误率也较高，这是因为拟合数据过于充分导致了过拟合。因此选择 minibatch=150 作为最终的参数。

- minibatch 为 50: 0.024000
- minibatch 为 100: 0.015000
- minibatch 为 150: 0.014000
- minibatch 为 200: 0.020000
- minibatch 为 500: 0.023000

4. 惩罚系数 λ：调整惩罚系数，查看对模型结果的影响。发现惩罚系数为 1e-2 时模型效果最好，可能是因为大于 1e-2 会让模型欠拟合，而小于 1e-2 会让模型过拟合风险增高，让模型最终的预测效果变差。

- 惩罚系数为 0: 0.021000
- 惩罚系数为 1e-3: 0.019000
- 惩罚系数为 5e-3: 0.017000
- 惩罚系数为 1e-2: 0.014000
- 惩罚系数为 5e-2: 0.022000

将结果最好的模型参数保存到本地。

[ ]:
```
inputWeights, hiddenWeights, outputWeights = W
savemat('data/bestmodel2.mat', {'inputWeights': inputWeights,
                                'hiddenWeights': hiddenWeights,
                                'outputWeights': outputWeights,
                                'penalty': 1e-2,
                                'alpha': 1e-4,
                                'minibatch': 150,
                                'inputFeatures': 256,
                                'nHidden': 16})
```

综上所述，我使用双层全连接神经网络，第一层有 256 个节点，第二层 16 个节点，学习率为 1e-4，惩罚系数为 1e-2，minibatch 为 150，对权重进行 HE 初始化，使用 RELU 神经元、Adam 优化器，使用 softmax 作为输出层的激活函数，在增强过后的数据集上进行训练，训练结束后使用 final tuning 进行微调，最终得到的错误率为 1.4%，即测试集上的分类准确率高达 98.6%。