# 常见优化器对比

何馨毅
西安交通大学电信学部，710049，西安

摘要（Abstract）： 优化算法在深度学习领域得到广泛应用，本文在课程基础上，进一步对深度学习领域最常见的优化器进行原理探究、算法实现探究、优化效果实验。调研发现，已有工作中对于各大优化器的对比，存在不全面、对原理证明不充分、缺少在预训练大模型上对比等问题，本文重点完善以上三点。本文首先选取3大类优化器（SGD、Adagrad、Adam）共9个优化器，进行深入探究，包括其原理、证明以及算法实现等。随后进行文本情感分类任务实验，在 GLUE-SST2数据集上采用 BERT 预训练模型，采用不同优化器对模型参数进行优化。最后对比优化过程中 Loss 变化、平均优化时间以及测试准确率等。实验发现，针对 NLP 预训练模型，Adagrad、Adam、AdamW、Adamax 优化器具有较好的优化效果，Adagrad 效率最高，AdamW 准确率最高。源代码见 https://github.com/Betty1202/optimizer_comparison.git。

关键词（Key words）： 深度学习，优化器，GLUE-SST2数据集，BERT

中图分类号：O121.8;G558

优化算法是我们在各项研究中所必须、最为基本的内容。在课堂中，教授们讲解了常见非线性优化方法，在深度学习领域也得到了广泛的应用。比如目前深度学习中最基本的 SGD 优化器便是最优化求解的迭代算法，Adadelta 优化器是牛顿迭代法的扩展等。本文将在课程基础上，进一步对深度学习领域最常见的优化器进行原理探究、算法实现探究、优化效果实验。调研发现，已有工作中对于各大优化器的对比，存在不全面、对原理证明不充分、缺少在预训练大模型上对比等问题，本文重点完善以上三点。

## 1 常见优化器原理

最优化问题是计算数学中最为重要的研究方向之一。而在深度学习领域，优化算法的选择也是一个模型的重中之重。即使在数据集和模型架构完全相同的情况下，采用不同的优化算法，也很可能导致截然不同的训练效果。梯度下降是目前神经网络中使用最为广泛的优化算法之一。为了弥补朴素梯度下降的种种缺陷，研究者们发明了一系列变种算法。下面我们深入对常见优化器原理进行探究。

设待优化的模型参数$\theta\epsilon\mathbb{R}^d$，目标函数（损失函数）$J(\theta)$，目标函数关于参数的梯度：

$$g_t = \nabla_\theta J(\theta)$$

本文将优化器分为 3 大类——SGD、Adagrad、Adam，并分别介绍其变体。三大类亦是承接关系，即 Adagrad 是 SGD 的改进，Adam 是 Adagrad 的改进版本。各优化器 pytorch 版本官方说明见[8]（中文）[9]（英文）。

### 1.1 SGD

#### 1.1.1 SGD

SGD 全称 Stochastic Gradient Descent，即随机梯度下降法，是最为简单基本的梯度下降法。其更新方向为梯度的反方向，更新公式如下：

$$\theta_{t+1} = \theta_t - \eta g_t$$

其中，$\eta$为学习率。

基本策略可以理解为随机梯度下降像是一个盲人下山，不用每走一步计算一次梯度，但是他总能下到山底，只不过过程会显得扭扭曲曲。其优点是虽然 SGD 需要走很多步的样子，但是对梯度的要求很低（计算梯度快）。而对于引入噪声，大量的理论和实践工作证明，只要噪声不是特别大，SGD 都能很好地收敛。应用大型数据集时，训练速度很快。缺点是收敛速度慢，可能在鞍点处震荡。并且，如何合理的选择学习率是 SGD 的一大难点。

Momentum 与 Nesterov 便是对于 SGD 的改进算法，下面对二者进行介绍。

收稿日期：2021-12-01. 作者简介：何馨毅（1999-），女，博士生，2174121271

### 1.1.2 Momentum

使用动量(Momentum)的随机梯度下降法(SGD)，主要思想是引入一个积攒历史梯度信息动量来加速 SGD。更新公式如下：

$$\begin{cases} m_t = \gamma m_{t-1} + \eta g_t \\ \theta_{t+1} = \theta_t - m_t \end{cases}$$

其中，$\gamma$为动力大小，$m_t$为 t 时刻积攒的动量。

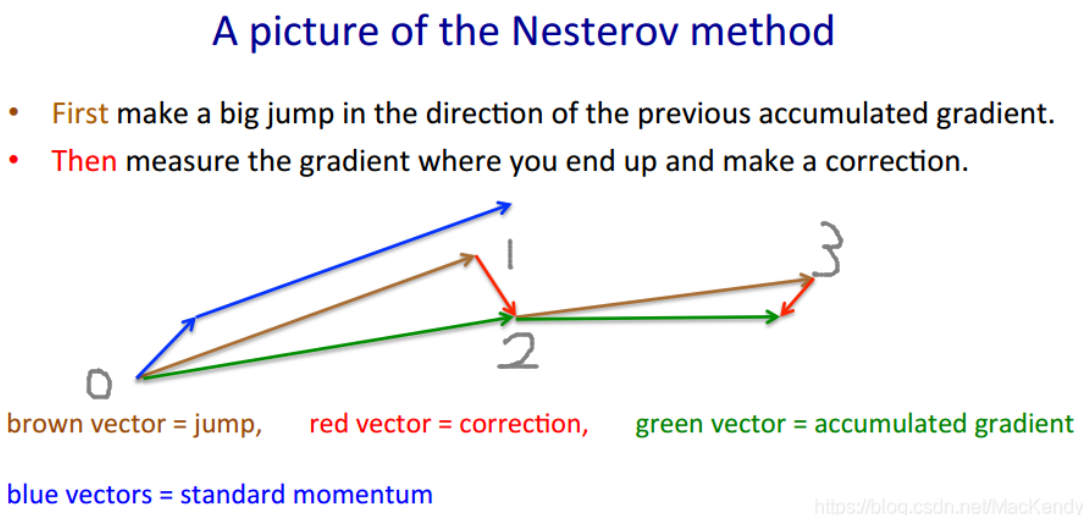
参数更新方向不仅由当前的梯度决定，也与此前累积的下降方向有关。这使得参数中那些梯度方向变化不大的维度可以加速更新，并减少梯度方向变化较大的维度上的更新幅度。由此产生了加速收敛和减小震荡的效果。

### 1.1.3 Nesterov

Nesterov Accelerated Gradient 在 Momentum 加入动量的基础上，增加先验知识，使得在得知快到最优解时能适当降低更新幅度，适应性会更好。更新公式如下所示：

$$\begin{cases} m_t = \gamma m_{t-1} + \eta \nabla_\theta J(\theta_t - \gamma m_{t-1}) \\ \theta_{t+1} = \theta_t - m_t \end{cases}$$

与 Momentum 的主要差别在于，梯度将$\theta_t$替换为$\theta_t - \gamma m_{t-1}$，用后者近似当作更新参数后的梯度，作为下一次梯度大小的先验知识。使用 Geoffrey Hinton 的例子，如图表 1 所示。Momentum 的步长计算了当前梯度（短蓝向量）和动量项 （长蓝向量）。然而，既然已经利用了动量项来更新 ，那不妨先计算出下一时刻的近似位置 （棕向量），并根据该未来位置计算梯度（红向量），然后使用和 Momentum 中相同的方式计算步长（绿向量），即 0->1->0->2。这种计算梯度的方式可以使算法更好的预测未来，提前调整更新速率。




图表 1 Nesterov 更新图

在实验中发现，Nesterov pytorch 版本的实现与原公式不同。Nesterov 与 Momentum 源码的不同，仅在于 Momentum 中动量更新为

$$\begin{cases} \text{buf} = \gamma buf_{t-1} + \nabla_\theta J \\ m_t = \text{buf} \end{cases}$$

而 Nesterov 中动量更新为

$$\begin{cases} \text{buf} = \gamma buf_{t-1} + \nabla_\theta J \\ m_t = \gamma \text{buf} + \nabla_\theta J \end{cases}$$

参考[4, 5]，下给出证明，根据 pytorch 中动量更新可推导出参数更新为

$$\begin{aligned} \theta_{t+1} &= \theta_t - m_t \\ &= \theta_t - \eta * (\nabla_\theta J + \gamma * (\gamma m_{t-1} + \nabla_\theta J)) \\ &= \theta_t - \eta * \nabla_\theta J - \eta * \gamma * (\gamma m_{t-1} + \nabla_\theta J) \\ &= \theta_t + \gamma m_{t-1} - (\gamma m_{t-1} + \eta * \nabla_\theta J) - \eta * \gamma * (\gamma m_{t-1} + \nabla_\theta J) \end{aligned}$$

Nesterov 的主要思想为提前走一步，再用这一步的梯度来更新参数。当前 step 的“提前走一步”其实就是上一 step 的“走多一步”，那不如每个 step 都走多一步，那下一个 step 就不用先试探性走，再更新参数了，即起始点变为 1，更新路径为 1->2->3。结合上式最终结果与图表 1：

$\theta_t + \gamma m_{t-1}$为提前走一步退回原始起点，即 1->0；

$\theta_t + \gamma m_{t-1} - (\gamma m_{t-1} + \eta * \nabla_\theta J)$为更新参数 1->0->2，即 1->2；

$\theta_t + \gamma m_{t-1} - (\gamma m_{t-1} + \eta * \nabla_\theta J) - \eta * \gamma * (\gamma m_{t-1} + \nabla_\theta J)$为更新参数并为下一步做准备，即 1->2->3。

Pytorch 版本与原 Nesterov 思想相同，但是优化了运算步骤。

## 1.2 Adagrad

### 1.2.1 Adagrad

Adagrad 是一种自适应优化方法，是自适应的为各个参数分配不同的学习率。这个学习率的变化，会受到梯度的大小和迭代次数的影响。深度学习模型中往往涉及大量的参数，不同参数的更新频率往往有所区别。对于更新不频繁的参数（典型例子：更新 word embedding 中的低频词），我们希望单次步长更大，多学习一些知识；对于更新频繁的参数，我们则希望步长较小，使得学习到的参数更稳定，不至于被单个样本影响太多。Adagrad 算法引入二阶动量实现此效果，其参数更新公式如下：

$$\begin{cases} v_t = diag\left(\sum_{i=1}^{t} g_{i,1}^2, \sum_{i=1}^{t} g_{i,2}^2, \cdots, \sum_{i=1}^{t} g_{i,d}^2\right) = v_{t-1} + diag(g_t^2) \\ \theta_{t+1} = \theta_t - \dfrac{\eta}{\sqrt{v_t + \epsilon}} g_t \end{cases}$$

对于此前频繁更新过的参数，其二阶动量的对应分量较大，学习率就较小。这一方法在稀疏数据的场景下表现很好。缺点是训练后期，学习率过小，因为 Adagrad 累加之前所有的梯度平方作为分母。

### 1.2.2 RMSprop

在 Adagrad 中，二阶动量$v_t$是单调递增的，使得学习率逐渐递减至 0，可能导致训练过程提前结束。为了改进这一缺点，可以考虑在计算二阶动量时不累积全部历史梯度，而只关注最近某一时间窗口内的下降梯度。其关于二阶动量$v_t$的更新公式为

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

其二阶动量采用指数移动平均公式计算，这样即可避免二阶动量持续累积的问题。

### 1.2.3 Adadelta

在 RMSprop 基础上，为动态改变学习率参数$\eta$，以及进一步加快更新速度，Adadelta 引入状态变量$\Delta x_t$，并用其均方根代替学习率参数$\eta$。其更新如下：

$$\begin{cases} v_t = \gamma v_{t-1} + (1-\gamma) diag(g_t^2) \\ u_t = \gamma u_{t-1} + (1-\gamma) diag(\Delta x_{t-1}^2) \\ \Delta x_t = \dfrac{\sqrt{u_t + \epsilon}}{\sqrt{v_t + \epsilon}} g_t \\ \theta_{t+1} = \theta_t - \Delta x_t \end{cases}$$

该算法可用牛顿迭代法进行证明。牛顿迭代法中二阶牛顿迭代公式为：

$$x_{t+1} = x_t - \frac{1}{f''(x)} * f'(x)$$

由上式可知，高阶牛顿迭代法迭代步长是 Hessian 矩阵。而 Adadelta 算法正是采用了 Hessian 矩阵的对角线进行近似，公式如下：

$$\Delta x \approx \frac{\frac{\partial f}{\partial x}}{\frac{\partial^2 f}{\partial x^2}}$$

$$\Rightarrow \quad \frac{1}{\frac{\partial^2 f}{\partial x^2}} = \frac{\Delta x}{\frac{\partial f}{\partial x}}$$

$$\Rightarrow \quad x_{t+1} = x_t - \frac{\Delta x}{\frac{\partial f}{\partial x}} * g(x)$$

对于迭代步长分子分母取均方根，得 Adadelta 更新公式。

### 1.3 Adam

#### 1.3.1 Adam

Adam 可以认为是 RMSprop 和 Momentum 的结合。和 RMSprop 对二阶动量使用指数移动平均类似，Adam 中对一阶动量也是用指数移动平均计算；并且对一阶和二阶动量做偏置校正。

$$\begin{cases} m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t = \beta_2 v_{t-1} + (1-\beta_2) diag(g_t^2) \\ \widehat{m_t} = \frac{m_t}{1-\beta_1}, \widehat{v_t} = \frac{v_t}{1-\beta_2} \\ \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \widehat{m_t} \end{cases}$$

其中，第三行为偏执矫正。在迭代初始阶段，$m_t$和$v_t$有一个向初值的偏移（过多的偏向了 0）。因此，可以对一阶和二阶动量做偏置校正 (bias correction)。算法保证迭代较为平稳。

#### 1.3.2 AdamW

Adam 虽然收敛速度快，但没能解决参数过拟合的问题。学术界讨论了诸多方案，其中包括在损失函数中引入参数的 L2 正则项（即$\text{Loss} = \text{Loss} + \lambda\theta_t^2$，算法实现中体现为$g_t = g_t + \lambda\theta_t$）。这样的方法在其他的优化器中或许有效，但会因为 Adam 中自适应学习率的存在而对使用 Adam 优化器的模型失效。AdamW 的出现便是为了解决这一问题，达到同样使参数接近于 0 的目的。具体的举措，是在 Adam 基础上，在最终的参数更新时引入参数自身：

$$\theta_{t+1} = \theta_t - \eta(\frac{\widehat{m_t}}{\sqrt{v_t + \epsilon}} + \lambda * \theta_t)$$

这一优化策略目前正广泛应用于各大预训练语言模型。

#### 1.3.3 Adamax

在 Adam 算法中，对于二阶动量的计算采用 2 阶范数，Adamax 将其扩展到 p 阶范数。对于较大的 p，这样的变量在数值上是不稳定的。然而，在我们让$p \to \infty$的特殊情况下，一个简单而稳定的算法出现了：

$$\begin{cases} u_t = \max(\beta_2 u_{t-1}, |g_t|) \\ \theta_{t+1} = \theta_t - \frac{\eta}{u_t} \widehat{m_t} \end{cases}$$

其证明[6]如下：

$$\begin{aligned} v_t &= \beta_2^p v_{t-1} + (1-\beta_2^p)|g_t|^p \\ &= (1-\beta_2^p) \sum_{i=1}^{t} \beta_2^{p(t-i)} |g_i|^p \end{aligned}$$

对 p 求极限，可得$u_t$如下：

$$
\begin{aligned}
u_t = \lim_{p\to\infty}(v_t)^{1/p} &= \lim_{p\to\infty}\left(\left(1-\beta_2^p\right)\sum_{i=1}^{t}\beta_2^{p(t-i)}|g_i|^p\right)^{1/p} \\
&= \lim_{p\to\infty}\left(\sum_{i=1}^{t}\left(\beta_2^{(t-i)}|g_i|\right)^p\right)^{1/p} \\
&= \max\left(\beta_2^{(t-1)}|g_1|,\beta_2^{(t-2)}|g_2|,\ldots,|g_t|\right) \\
&= \max(\beta_2 u_{t-1},|g_t|)
\end{aligned}
$$

# 2　优化任务

随着研究机器学习的科研人员愈来愈多，已有工作中不乏对于现有常见优化器的对比。[1]选取了最常见的四种优化器进行算法对比，并在简单网络上进行优化对比。[2]主要对比了当时（2019 年 05 月）pytorch 中 10 个优化器的参数及功能。前者对于算法本身对比较为仔细，但是优化器选取不够全面，对比的网络过于简单，并未在当下广泛使用的预训练大模型上进行对比。后者虽然优化器选取较为全面，但是对比仅局限于参数及功能，对于原理以及真实实验结果涉及甚少。基于此，本文针对自然语言预训练模型，采用不同优化器进行 fine tune，观察优化器表现。本文选择自然语言处理中经典任务——情感分类任务，采用经典数据集——GLUE SST-2[3]，以及预训练模型 BERT。此任务的模型排行榜见[7]。

SST(Stanford Sentiment Treebank) [3] 数据集是一个具有完全标记的解析树语料库，可以对语言中情感的构成效应进行完整的分析。该语料库基于 GLUE 数据集，由 11,855 个单一句子组成，它们从电影评论中提取而来。数据集由 Stanford Parser 进行解析，其中包括来自 Stanford Parser 的总共 215,154 个独特的短语，每个短语均由 3 位人类评委进行注释。每个短语都被标记为负面、部分负面、中立、部分正面或正面。带有所有 5 个标签的语料库被称为 SST-5 或 SST 细粒度语料库。SST-2 是对完整句子的二元分类实验的数据集（负面或有点负面与有点正面或正面，中性句子被丢弃）。本文采用上述 SST-2 数据集。

BERT 是谷歌在 2018 年 10 月推出的深度语言表示模型，在 11 个 NLP 任务上的表现刷新了记录，包括问答 Question Answering (SQuAD v1.1)，推理 Natural Language Inference (MNLI) 等。其全称是 Bidirectional Encoder Representation from Transformers，即双向 Transformer 的 Encoder。模型的主要创新点都在 pre-train 方法上，即用了 Masked LM 和 Next Sentence Prediction 两种方法分别捕捉词语和句子级别的表征。MLM 在将单词序列输入给 BERT 之前，每个序列中有 15% 的单词被 [MASK] token 替换。 然后模型尝试基于序列中其他未被 mask 的单词的上下文来预测被掩盖的原单词。NSP 在 BERT 的训练过程中，模型接收成对的句子作为输入，并且预测其中第二个句子是否在原始文档中也是后续句子。在训练期间，50% 的输入对在原始文档中是前后关系，另外 50% 中是从语料库中随机组成的，并且是与第一句断开的。本文采用预训练的 base Bert 模型，连接线性层 fine tune，进行分类任务预测。

# 3　实验与结果

## 3.1　实验细节

实验主要分为数据处理、模型加载、训练与测试三个部分，实验入口主函数见 main.py，下面分别对三个步骤进行细节说明：

### 3.1.1　数据处理

数据处理部分代码详见 data.py 中函数 get_data()。首先加载数据，本文采用 HuggingFace 所提供 load_dataset()函数之间进行数据加载。其次对每条数据进行分词处理，本文采用 bert-base-uncased 对应分词器，对每条数据进行分词。随后采样数据集，由于实验重点在于优化器的对比，为加快实验进程，本实验对数据集进行采样进行实验，训练样本个数通过命令行参数 dataset_size 控制，测试样本个数为训练的十分之一。最后划分 batch，采用 DataLoader 进行 batch 划分，经过测试发现 GPU 显存可接受最大 batch 为 16，实验中将 batch_size 固定为 16。

#### 3.1.2 模型加载

本文采用 HuggingFace 提供的 BertForSequenceClassification 模型，尺寸为 bert-base-uncased。其中，Bert 部分参数已经过预训练，本实验中只对其进行 fine tune；分类任务线性层参数为初始化参数，本实验中对其进行训练。详细代码见 main.py。

#### 3.1.3 训练与测试

详细代码见 trainer.py 中 Trainer 类，其中 train()函数进行一次训练，epoch 个数通过命令行参数 epoch 控制，每个 epoch 对所有数据进行一次训练；test()函数进行一次测试，只进行一个 epoch，即对所有数据进行一次测试；iteration()函数为一次 epoch 中，对不同 batch 一次进行训练，经历前向传播、Loss 计算、梯度回传等步骤。

实验中优化器选择通过命令行参数 optimizer 控制，可选择优化器为 pytorch 中常见优化器——SGD、Momentum、Nesterov、Adagrad、RMSprop、Adadelta、Adam、AdamW、Adamax，详见 optimizer.py。实验中学习率采用线性学习率，即根据实验总共 step 数以及学习率初始值，对其进行线性插值，使得训练过程中学习率逐步递减。损失值 Loss 计算采用 BertForSequenceClassification 模型中分类任务损失值。

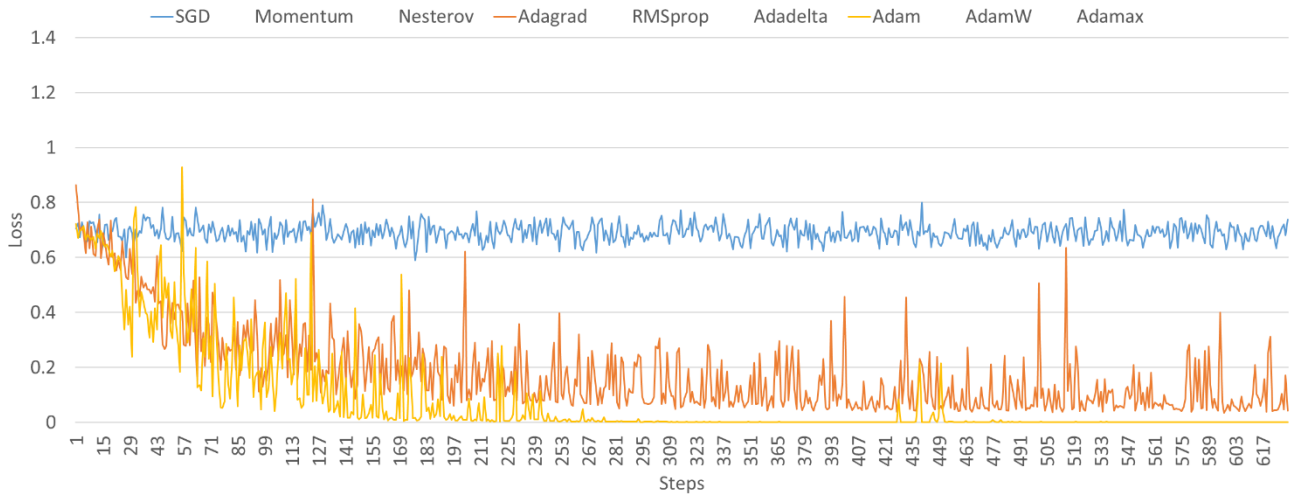
#### 3.1.4 模型与优化器评估

详细代码见 evaluation.py 中 Evaluation 类，其中记录训练或测试过程中，每个 epoch 以及每个 iteration 的优化时间、loss，以及每个 epoch 的准确率。函数 update_iter()在每个 iteration 后进行更新，更新度量值 metric，记录 loss 与优化时间。函数 update_epoch()在每个 epoch 后进行更新，根据每个 iteration 记录的度量值计算准确率，计算 epoch 中每个 iteration 平均优化时间以及 loss。函数 save()对所有记录的评估指标进行保存，得到 xxx_iter.xlsx 记录每个 iteration 的优化时间以及损失值 loss，xxx_epoch.xlsx 记录每个 epoch 的准确率以及平均 iteration 优化时间与 loss。

### 3.2 实验结果

#### 3.2.1 Loss 对比

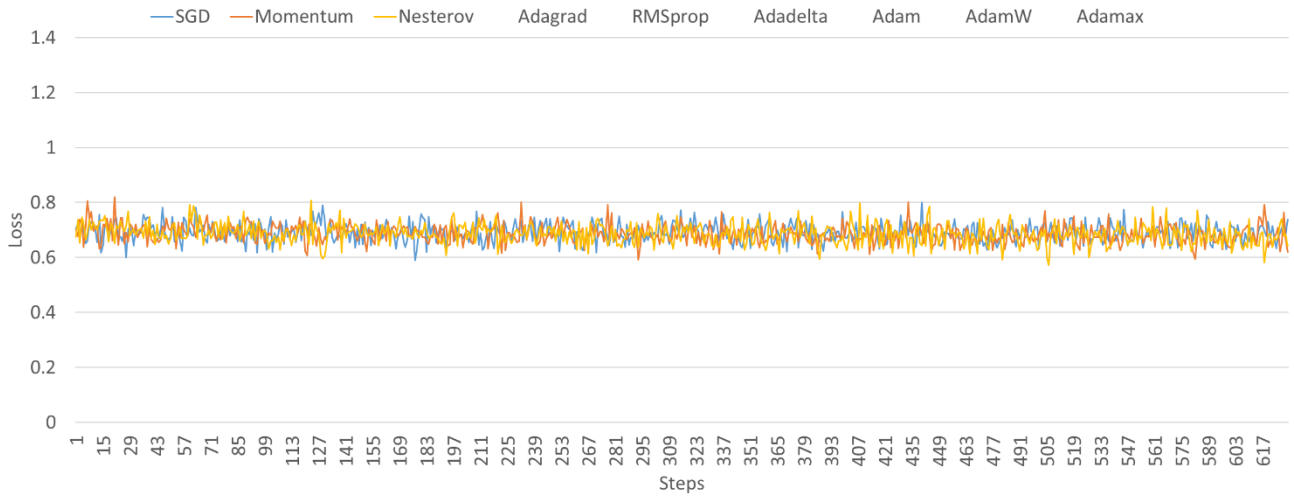
首先对比三大基本优化器——SGD、Adagrad、Adam 在训练过程中 Loss 变化，如图表 2 所示：




**图表 2 三大基本优化器 Loss 对比**

对比发现，最朴素的 SGD 优化器 Loss 基本未降低，可能对于高纬度嵌入易陷入局部最优点，致使无法找到参数空间最优点。Adagrad 的 Loss 有明显且稳定的降低，最终收敛于较小的 Loss，达到一个不错的效果。说明二阶动量对于学习率的改进，使得面对高纬度嵌入空间时，优化效果大大提升。Adam 的 Loss 降低最快，且最终收敛 Loss 值最低，达到三个优化器中最优的效果。

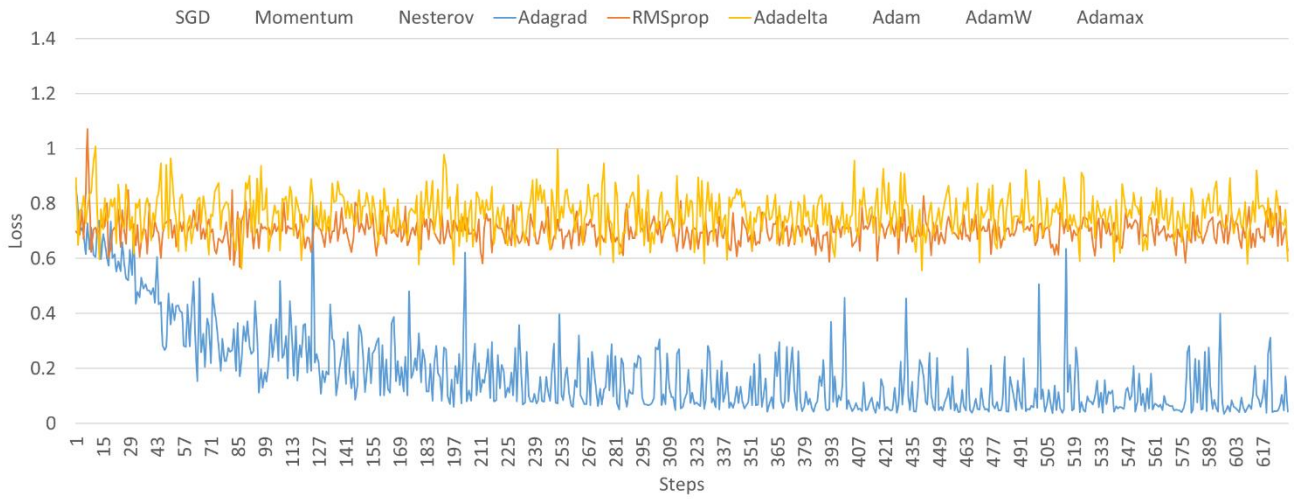
对比 SGD 及其变体（Momentum、Nesterov）优化器如图表 3 所示：




**图表** 3 SGD **及其变体优化器** Loss **对比**

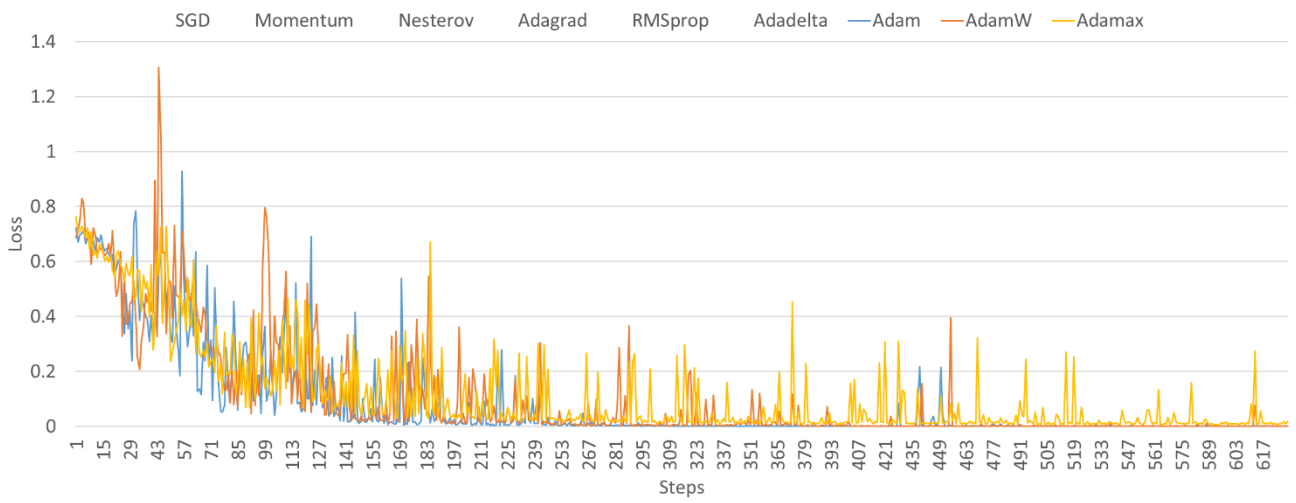
三者均未得到较好的优化，Loss 并未得到优化而下降，由于三者均未引入二阶动量对学习率进行修正，所以各个维度的学习步长相同，对于 NLP 高纬度嵌入问题，易陷入局部最优点，致使 Loss 不能得到充分优化。

对比 Adagrad 及其变体（RMSprop、Adadelta）优化器如图表 4 所示：




**图表** 4 Adagrad **及其变体优化器** Loss **对比**

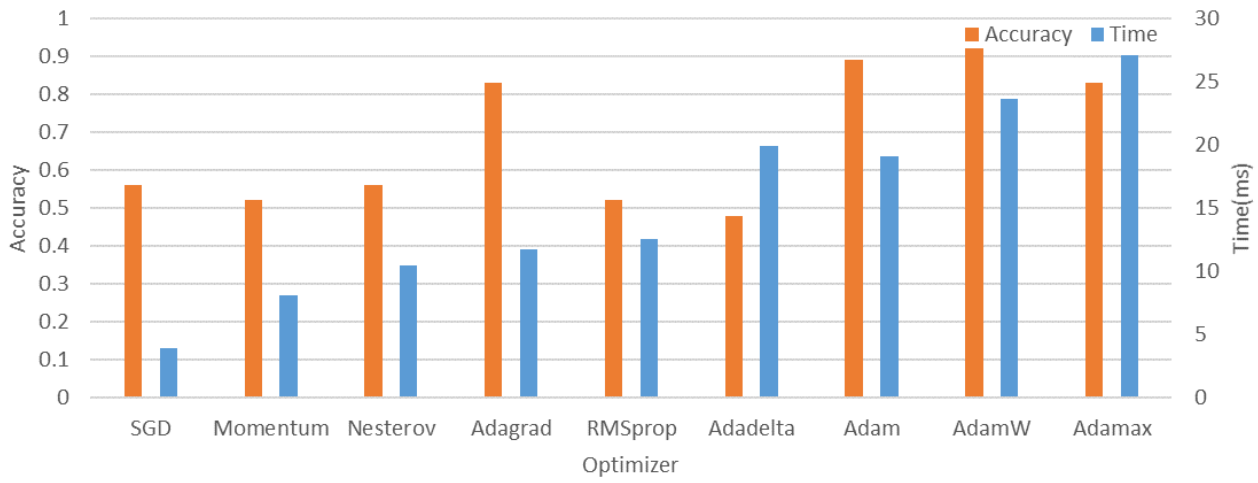
对比发现，三者只有 Adagrad 的 Loss 得到了较好的优化，而 RMSprop 与 Adadelta 的 Loss 基本未得到优化。由于二者对于二阶动量仅累计部分历史二阶动量，在起始时有一个向初值的偏移，并未进行偏置矫正，致使初始优化效果不佳，如图表 4 中 1-15steps，二者 Loss 产生骤升。




**图表** 5 Adam **及其变体优化器** Loss **对比**

三者优化效果总体较优。其中，Adamax 最终稳定性不如另两者，由于其取高阶范数，使得对于扰动过于敏感，稳定性相对较差。AdamW 在 Loss 下降阶段不如另两者平滑（如 steps43-75），由于其为避免过拟合，在参数更新阶段引入参数本身，虽然很好避免了过拟合，但是对于优化时的平滑性下降。

### 3.2.2 计算时间与准确率对比




**图表** 6 **各优化器训练集优化时间与测试集准确率对比**

由图表 6 可知，针对优化时间，随着优化器的复杂程度增加而增加，SGD 类优化时间最短，其次是 Adagrad 类优化器，时间最长的是 Adam 类优化器。针对准确率，SGD、Momentum、Nesterov、RMSprop、Adadelta 优化器准确率为 50%左右，由于任务为二分类任务，随机分准确率为 50%，故可见 5 个优化器对于模型基本未优化，与 3.2.1 节 Loss 优化图结论相符。

针对准确率达到 0.8 以上的 Adagrad、Adam、AdamW、Adamax 优化器，Adagrad 用时最短，AdamW 准确率最高，这也是 BERT 在预训练是所选择的优化器。AdamW 与 Adam 的 Loss 曲线近似，但是准确率前者更高，因为 AdamW 可以更好避免过拟合现象，故即使在训练集上最终 Loss 相近，但是在测试集上会有不同的效果。针对 NLP 预训练模型，对于不同优化器的选择，还要看具体应用场景，是更追求效率，还是更追求准确率。

## 4 结论

本文分析对比常见的 9 大优化器，发现针对 NLP 预训练模型，Adagrad、Adam、AdamW、Adamax 优化

器具有较好的优化效果，Adagrad 效率最高，AdamW 准确率最高，对于不同优化器的选择，还要看具体应用场景，是更追求效率，还是更追求准确率。

**参考文献：**

[1] TensorSense. PyTorch 学习笔记（七）：PyTorch 的十个优化器[EB/OL]. [2021/11/28]. https://blog.csdn.net/u011995719/article/details/88988420

[2] ShuYini. Pytorch 中常用的四种优化器 SGD、Momentum、RMSProp、Adam[EB/OL]. [2021/11/28]. https://zhuanlan.zhihu.com/p/78622301

[3] Socher R, Perelygin A, Wu J, et al. Recursive deep models for semantic compositionality over a sentiment treebank[C]//Proceedings of the 2013 conference on empirical methods in natural language processing. 2013: 1631-1642.

[4] dontloo. [EB/OL]. [2021/11/28]. https://stats.stackexchange.com/questions/179915/whats-the-difference-between-momentum-based-gradient-descent-and-nesterovs-acc/191727#191727

[5] kendyChina. 基于 Pytorch 源码对 SGD、momentum、Nesterov 学习[EB/OL]. [2021/11/28]. https://blog.csdn.net/MacKendy/article/details/106742961

[6] Kingma D P, Ba J. Adam: A method for stochastic optimization[J]. arXiv preprint arXiv:1412.6980, 2014.

[7] Sentiment Analysis on SST-2 Binary classification[EB/OL]. [2021/11/28]. https://paperswithcode.com/sota/sentiment-analysis-on-sst-2-binary

[8] torch.optim[EB/OL]. [2021/11/28]. https://pytorch-cn.readthedocs.io/zh/latest/package_references/torch-optim/

[9] TORCH.OPTIM[EB/OL]. [2021/11/28]. https://pytorch.org/docs/stable/optim.html