دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

# گزارش ۸ درس هوش مصنوعی

# مقایسه روشهای مختلف فرا ابتکاری برای بهینه سازی ضرایب یک شبکه عصبی برای حل یک مساله طبقه بندی benchmark

به قلم:
امیر بابامحمودی

استاد
دکتر مهدی قطعی

خرداد ۱۴۰۰

## مقدمه:

در این گزارش قصد داریم از یک سری optimizer مختلف برای بررسی تغییراتی که در نتیجه یک مسئله ی طبقه بندی ایجاد میکنن استفاده کنیم.

## دیتاست titanic :

### لینک دریافت دیتاست:

https://www.kaggle.com/c/titanic/data

این دیتا ست شامل اطلاعات ۸۹۱ مسافر کشتی تایتانیک میباشد که با استفاده از این اطلاعات میخواهیم ببینیم که کدام یک از مسافران نجات یافته و کدام یک غرق شده اند.

| | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 0 | 3 | Braund, Mr. Owen Harris | male | 22.0 | 1 | 0 | A/5 21171 | 7.2500 | NaN | S |
| **2** | 3 | 1 | 3 | Heikkinen, Miss. Laina | female | 26.0 | 0 | 0 | STON/O2. 3101282 | 7.9250 | NaN | S |
| **3** | 4 | 1 | 1 | Futrelle, Mrs. Jacques Heath (Lily May Peel) | female | 35.0 | 1 | 0 | 113803 | 53.1000 | C123 | S |
| **4** | 5 | 0 | 3 | Allen, Mr. William Henry | male | 35.0 | 0 | 0 | 373450 | 8.0500 | NaN | S |
| **6** | 7 | 0 | 1 | McCarthy, Mr. Timothy J | male | 54.0 | 0 | 0 | 17463 | 51.8625 | E46 | S |

از اطلاعات موجود در جدول بالا نیاز میباشد که ستون survived رو به عنوان target جدا کرده و ستون های جنسیت, سن, SibSp که همان همسر و خواهر برادر مربوط به مسافر که سوار کشتی بوده است, Parch که پدر مادر و یا فرزند مسافر که مسافر کشتی باشند و همچنین لوکیشنی که سوار کشتی شدند یعنی embarked را برای train کردن استفاده کنیم.

```
gender = {'male': 1,'female': 2}
titanic.Sex = [gender[item] for item in titanic.Sex]
titanic = titanic[titanic['Embarked'].notna()]
Embark = {'S' : 1 , 'C' : 2 , 'Q' : 3}
titanic.Embarked = [Embark[item] for item in titanic.Embarked]
```

ابتدا ستون های شامل رشته رو به یک عدد کد میکنیم برای مثال جنسیت مرد ۱ و خانم ۲. و همچنین برای Embarked.

حال دو جدول متفاوت ساخته یکی برای داده های ورودی دیگری به عنوان لیبل که نجات یافته اند یا خیر.
سپس این دو جدول را به نسبت۸۰ به ۲۰ برای trainset و testset تقسیم میکنیم.که کد آن مانند شکل زیر میشود.

```
titanic_select = titanic[[ "Pclass", "Sex", "Age", "SibSp", "Parch", "Embarked"]].copy()
survived = titanic[["Survived"]]
```

```
x_train = titanic_select[:650]
y_train = survived[:650]
x_test = titanic[650:]
y_test = survived[650:]
```

## :MLP

اختصاری از Multilayer Perceptron شامل لایه هایی از نورون ها که با گرفتن ورودی و انجام دادن پردازش هایی روی آن پارامتر های مرتبط با هر لایه رو حساب کرده و در نهایت در لایه ی آخر به تعداد کلاس ها نورون وجود دارد که احتمال تعلق هر دیتا به این کلاس ها را میسنجد. از آنجایی که دیتای انتخابی در این گزارش بسیار سبک بوده و کم میباشد از تنها دو hidden layer استفاده شده است.
در لایه اول ابعاد ورودی که شامل تنها یک ارایه یک بعدی شامل ۶ feature میباشد را میدهیم و در لایه های hidden از تابع relu استفاده شده و در نهایت در لایه آخر که شامل دو کلاس زنده ماند یا مردن است از sigmoid استفاده میکنم.

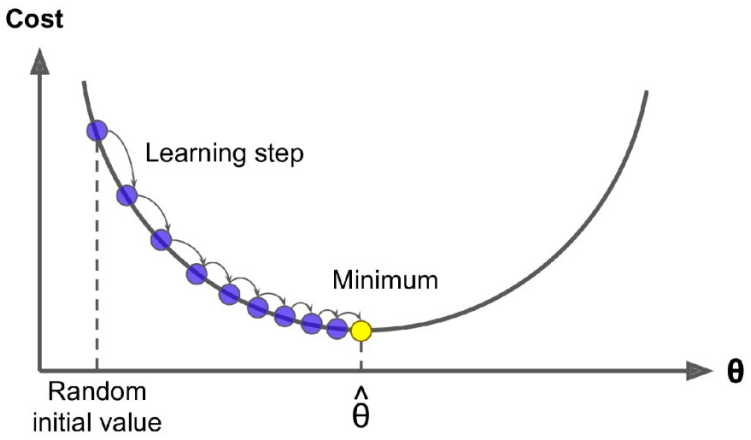
```
import tensorflow as tf
from tensorflow import keras
from sklearn.metrics import classification_report
import tensorboard
model = keras.models.Sequential([
keras.layers.InputLayer(input_shape = 6),
keras.layers.Dense(18, activation="relu"),
keras.layers.Dense(10, activation="relu"),
keras.layers.Dense(2, activation="sigmoid")])
model.save_weights("weights")
```

همانطور که میبینید در آخر کد از save_weights استفاده شده است. از آنجایی که در ابتدا پارامتر ها به صورت رندوم وزن داده میشوند و ممکن در هر بار ران کردن مدل با اوپتیمایزر های مختلف نتایج را نتوان در شرایط یکسان مقایسه کرد با ذخیره کردن وزن های رندوم اولیه و load کردن آن ها در هر مرتبه میتوانیم از این اتفاق خودداری کرده و تحلیل بهتری را روی نتایج انجام بدیم.

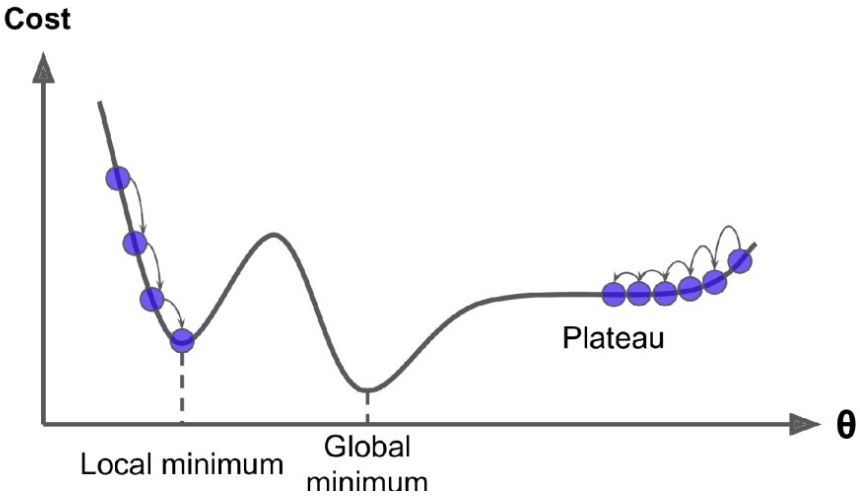
حال به پردازش این مدل بر روی دیتا ها با optimizer های مختلف میپردازیم و نتایج رو مقایسه میکنیم.

## :SGD

بطور کلی ایده اصلی gradient descent ها تغییر دادن جزیی مقادیر پارامتر ها در جهت شیب کاهش loss function میباشد. بدین گونه که در ابتدا به صورت رندوم و با توزیع نرمال وزن ها مقدار میگیرند و در حین back propogation تلاش میشود که با حرکت در جهت شیب و کمتر شدن loss function حرکت کرد.

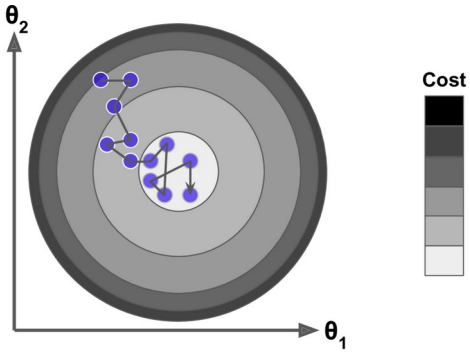



چیزی که در gradient descent ها هائز اهمیت میباشد طول گامی میباشد که برای تغییر وزن پارامتر ها انتخاب میکنیم که از آن بعنوان learning rate یاد میشود. حال با یک شکل به برخی مشکلات gradient descent ها میپردازیم.




همانگونه که میبینید ممکن است در مینیمم محلی گیر کرده و چون پس از آن دیگر loss function دوباره صعودی میشود همان نقطه به عنوان وزن های بهینه انتخاب شود در حالی که شاید اگر از نقطه ی رندوم

دیگری شروع میکردیم این مشکل پیش نمیامد. یا ممکن هست به فلات ها بربخوریم. مناطقی که شیب آن ها بسیار کم بوده و باعث میشود که از نظر زمانی خیلی الگوریتم ما بد بشود.
برای اینکه مشکلات مذکور کمتر رخ بدهند از تکنیکی تحت عنوان stochastic gradient descent یا همان SGD استفاده میکنیم. در این روش هربار به صورت رندوم یک دیتا برداشته شده و loss function تنها با توجه به خروجی این دیتا مشخص میشود. بدیهیست که این تکنیک سرعت الگوریتم رو بسیار افزایش میدهد زیرا دیگر مجبور نیستیم روی کل دیتا محاسباتی را انجام دهیم. مشکلی که این الگوریتم دارد این میباشد که هیچگاه متوقف نشده و همواره در نزدیکی نقطه بهینه ما هم ممکن است حرکت کند که با معیاری میتوان آن را متوقف کرد و این کار به ما این اطمینان را میدهد که به شدت به جواب مینیمم مطلق نزدیک شدیم و در مینیمم محلی گیر نکردیم. در شکل زیر نمایی از این الگوریتم را میبینیم.




حال به بررسی کد آن میپردازیم:

```
:  1 model.load_weights("weights")
   2 optimizer2 = keras.optimizers.SGD(lr=0.002)
   3 model.compile(loss="sparse_categorical_crossentropy",
   4 optimizer=optimizer2,
   5 metrics=["accuracy"])
   6 history["SGD"] = model.fit(x_train_scale , y_train , epochs=25 )
   7 predict_mlp = model.predict_classes(x_test_scale)
   8 print(classification_report(predict_mlp , y_test))
```

```
/home/amir/anaconda3/lib/python3.8/site-packages/tensorflow/python/keras/optimizer_v2/optim
ning: The `lr` argument is deprecated, use `learning_rate` instead.
  warnings.warn(

Epoch 1/25
17/17 [==============================] - 0s 781us/step - loss: 0.7132 - accuracy: 0.4755
Epoch 2/25
17/17 [==============================] - 0s 1ms/step - loss: 0.7120 - accuracy: 0.4849
Epoch 3/25
17/17 [==============================] - 0s 1ms/step - loss: 0.7108 - accuracy: 0.4962
Epoch 4/25
17/17 [==============================] - 0s 1ms/step - loss: 0.7096 - accuracy: 0.5019
Epoch 5/25
```

همانگونه که در کد میبینید در هنگام compile کردن کد آرگومانی تحت عنوان optimizer وجود دارد که در آن از SGD استفاده کردیم که در آن از learning_rate = 0.002 استفاده شده است.

## Momentum SGD:

در روش sgd معمولی در هر مرحله کاری به شیب تابع loss در مرحله ی قبل نداشته و شیب لحظه محاسبه شده و ضریبی از آن از وزن پارامتر کم میشود در روش momentum اینگونه در نظر میگیریم که انگار شتاب در لحظه پایین آمدن را شبیه سازی کرده و هر چه به نقطه ی مینیمم نزدیک تر میشویم سرعت حرکت به سمت آن بیشتر شده و از نظر زمانی بهتر از SGD میباشد. در اصل یعنی یک بردار تجمعی از شیب loss function در مرحله های قبل را محاسبه کرده سعی میکند از گرادیان مراحل قبل تاثیر بپذیرد. این نکته نیز هائز اهمیت میباشد که از یک ضریب B استفاده میکنیم تا بتوانیم شتاب را کنترل کرده و در اصل اصطکاک در هنگام لغزش را هم شبیه سازی کنیم که فرمول آن به شکل زیر در میاید:

1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}\right)$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$

که B بین ۰ و ۱ میباشد.

حال کد این بخش را میبینیم:

```
model.load_weights("weights")
optimizer2 = keras.optimizers.SGD(lr=0.002, momentum=0.9)
model.compile(loss="sparse_categorical_crossentropy",
optimizer=optimizer2,
metrics=["accuracy"])
history["momentum"] = model.fit(x_train_scale , y_train , epochs= 25 )
predict_mlp = model.predict_classes(x_test_scale)
print(classification_report(predict_mlp , y_test))
```

```
/home/amir/anaconda3/lib/python3.8/site-packages/tensorflow/python/keras/optimizer_v2/optimizer_v2.py:
ning: The `lr` argument is deprecated, use `learning_rate` instead.
  warnings.warn(
```

```
Epoch 1/25
17/17 [==============================] - 0s 936us/step - loss: 0.7116 - accuracy: 0.4868
Epoch 2/25
17/17 [==============================] - 0s 1ms/step - loss: 0.7033 - accuracy: 0.5566
Epoch 3/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6950 - accuracy: 0.5830
Epoch 4/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6881 - accuracy: 0.5887
Epoch 5/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6819 - accuracy: 0.5906
Epoch 6/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6768 - accuracy: 0.5906
Epoch 7/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6721 - accuracy: 0.5906
Epoch 8/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6678 - accuracy: 0.5906
Epoch 9/25
```

همانگونه که میبینید کد آن مانند حالت قبل بوده با این تفاوت که آرگومان momentum = 0.9 به آن داده شده که مقدار 0.9 یک مقدار معمول برای آن میباشد.

## :RMSProp

همانگونه که از ساز و کار روش های GD مشخص میباشد همواره تمایل به رفتن در جهتی دارند که تابع loss مارا کوچیک تر کنند اما گاها ممکن است این حرکت به سمت نقطه ی مینیمم نباشد پس نیاز میباشد که از حرکت در جهت عمود بر نقطه مینیمم خودداری شود. در اینجا دو روش وجود دارند AdaGrad که در آن برداری از مجموع مربع‌های بردار ها گرادیان محاسبه شده و در هر مرحله گرادیان جدید را بر جذر این بردار تقسیم کرده و اصلاح میکند که این کار باعث میشود جهت حرکت به سمت بیشترین شیب اصلاح شده باشد حال در روش RMSProp از تمامی گرادیان های قبلی استفاده نشده و تنها چند گرادیان آخر استفاده میشود.که فرمول محاسبه آن را نیز در مقابل میبینید.

$$\mathbf{s} \leftarrow \beta \mathbf{s} + (1-\beta)\, \nabla_{\theta} J(\boldsymbol{\theta}) \otimes \nabla_{\theta} J(\boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\, \nabla_{\theta} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$$

کد این روش را در زیر میبینید:

```
model.load_weights("weights")
optimizer1 = keras.optimizers.RMSprop(lr=0.002, rho=0.9)
model.compile(loss="sparse_categorical_crossentropy",
optimizer=optimizer1,
metrics=["accuracy"])
history["RMSprop"] = model.fit(x_train_scale , y_train , epochs=25 )
predict_mlp = model.predict_classes(x_test_scale)
print(classification_report(predict_mlp , y_test))
```

```
/home/amir/anaconda3/lib/python3.8/site-packages/tensorflow/python/keras/optimizer_v2/opt
ning: The `lr` argument is deprecated, use `learning_rate` instead.
  warnings.warn(
Epoch 1/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6868 - accuracy: 0.5491
Epoch 2/25
17/17 [==============================] - 0s 802us/step - loss: 0.6510 - accuracy: 0.5887
Epoch 3/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6232 - accuracy: 0.5906
Epoch 4/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6000 - accuracy: 0.5906
Epoch 5/25
17/17 [==============================] - 0s 1ms/step - loss: 0.5807 - accuracy: 0.6057
Epoch 6/25
17/17 [==============================] - 0s 1ms/step - loss: 0.5657 - accuracy: 0.7566
```

## Adam:

ترکیبی از دو روش momentum و RMSProp میباشد. یعنی هم در راستای افزایش شتاب حرکت سمت نقطه مینیمم و هم حرکت دقیق به سمت نقطه‌ی مینیمم اقدام میکند. که در ادامه فرمول های ریاضی آن را میبینید:

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

$$\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

$$\widehat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - {\beta_1}^{\mathsf{T}}}$$

$$\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - {\beta_2}^{\mathsf{T}}}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \widehat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \varepsilon}$$

کد این بخش:

```
model.load_weights("weights")
optimizer3 = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
model.compile(loss="sparse_categorical_crossentropy",
optimizer=optimizer3,
metrics=["accuracy"])
history["Adam"] = model.fit(x_train_scale , y_train , epochs=25 )
predict_mlp = model.predict_classes(x_test_scale)
print(classification_report(predict_mlp , y_test))
```

```
/home/amir/anaconda3/lib/python3.8/site-packages/tensorflow/python/keras/optimizer_v2/optimizer
ning: The `lr` argument is deprecated, use `learning_rate` instead.
  warnings.warn(
```

```
Epoch 1/25
17/17 [==============================] - 0s 922us/step - loss: 0.7061 - accuracy: 0.5132
Epoch 2/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6880 - accuracy: 0.5755
Epoch 3/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6733 - accuracy: 0.5887
Epoch 4/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6584 - accuracy: 0.5925
Epoch 5/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6438 - accuracy: 0.5906
Epoch 6/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6292 - accuracy: 0.5906
Epoch 7/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6142 - accuracy: 0.5906
Epoch 8/25
17/17 [==============================] - 0s 1ms/step - loss: 0.6003 - accuracy: 0.5906
```

حال به شرح معیار هایی که برای مقایسه سه مدل پیاده سازی شده استفاده شده است میپردازیم.

## :Precision

$$\text{precision} = \frac{TP}{TP + FP}$$

که TP تعداد داده های مثبتی که درست حدس زده شده اند و FP داده هایی که به اشتباه درست حدس زده شده اند میباشد.(یعنی زنده تشخیص داده شده اند)

## :recall

$$\text{recall} = \frac{TP}{TP + FN}$$

FN تعداد داده های منفی ای که به اشتباه منفی حدس زده شده اند.(در اینجا منفی یعنی مرده ها)

## :F1-Score

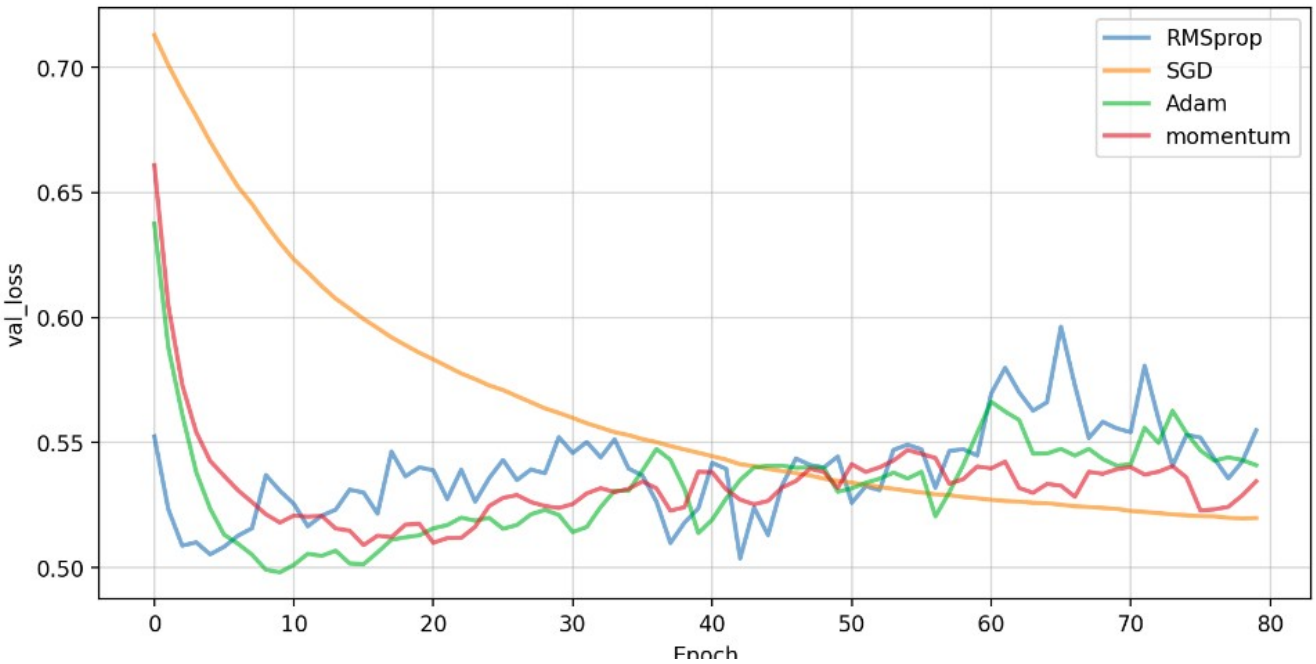
ترکیبی از دو فرمول ذکر شده در بالا میباشد که به آن میانگین موزون(harmonic mean) هم گفته میشود.و دلیل این نام گذاری این میباشد که به مقادیر کمتر ارزش بیشتری میدهد.که این بدان معنی میباشد که این معیار زمانی مقدارش زیاد میشود که هر دو معیار دیگر مقدار زیادی داشته باشند.

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

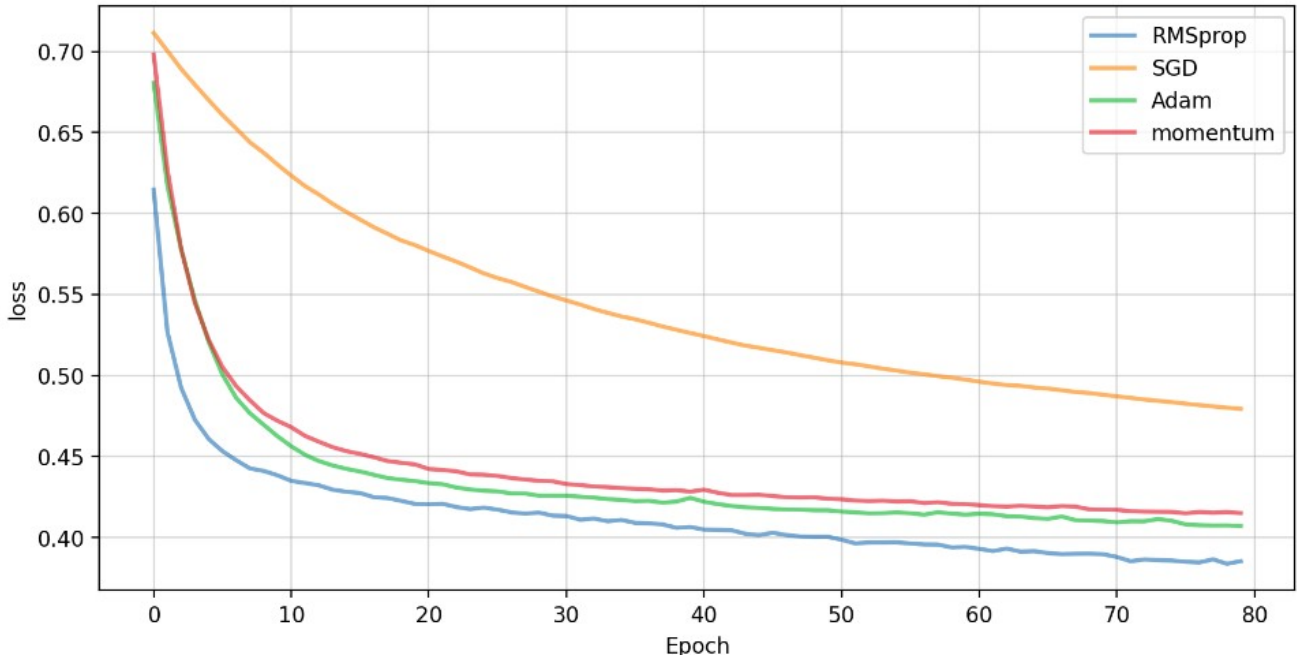
حال به مقایسه خروجی هر سه روش با این سه معیار میپردازیم:

| F1-Score | precision | recall | معیار/optimizer |
|---|---|---|---|
| 0.82 | 0.83 | 0.82 | SGD |
| 0.84 | 0.85 | 0.84 | SGD with momentum |
| 0.85 | 0.85 | 0.85 | RMSProp |
| 0.84 | 0.85 | 0.84 | Adam |

حال به تحلیل نموداری loss و accuracy مدل های مختلف میپردازیم در epoch = 25 میپردازیم:




همانگونه که میبینید در روش های RMSProp, Adam و momentum در ابتدا validation loss بسیار سریع کاهش یافته سپس در حال زیاد شدن میباشد که ممکن هست نشان دهنده‌ی خطر overfit باشد که میتواند با روش‌های مختلف regularization از آن جلوگیری نمود و به طور کلی میبینید که روش SGF کند ترین سرعت کاهش را دارد اما همواره در حال بهتر شدن میباشد زیرا سعی در پیدا کردن نقطه مینیمم است و این نشان میدهد که در ۹۰ ایپاک همچنان به نقطه مینیمم بسیار نزدیک نشده که در حال نوسان بیوفتد اما سه الگوریتم دیگر اینگونه شده اند بخاطر سرعت بسیار بهتری که دارند.




همانگونه که انتظار میرفت SGD سرعت پایین تری نسبت به بقیه الگوریتم در کاهش loss داشته و بر روی این دیتاست RMSProp با سرعت بهتریبه نتیجه‌ی مطلوب رسیده است.

در نهایت لازم به ذکر است که optimizer ها تاثیر به سزایی در بهبود سرعت الگوریتم داشته و میبینیم وقتی با epoch = 80 مدل هارو train میکنیم تقریبا دقت در در همه‌ی آپتیمایزر ها شبیه به هم و حدود 0.84 شد که میتوان گفت در نهایت در میزان دقت شاید تاثیر کمتری از جانب آپتیمایزر ها ببینیم.

## منابع:


۱ـ hands-on machine learning with scikit-learn and tensorflow aurelien geron

۲ـ https://towardsdatascience.com/a-look-at-gradient-descent-and-rmsprop-optimizers-f77d483ef08b

3ـ گزارش شماره‌ی ۷ خودم که لینک دسترسی آن را در زیر میبینید:
https://github.com/amirbabamahmoudi/AI-projects/tree/main/classification_compare_models

لینک گیتهاب جهت دسترسی به کد:
https://github.com/amirbabamahmoudi/AI-projects/tree/main/optimizer_cpmparison