Evaluation of Gradient Descent Optimization Techniques in Deep Neural Networks

Arun Vignesh Malarkkan(1215098183), Balaji Gokulakrishnan(1215173401), Gowtham Sekkilar(1215181396), Raghavendran Ramakrishnan(1215325696)

***Abstract*—The goal of this paper is to evaluate the performance of the various gradient descent optimization techniques by training a Deep Neural Network. The performance is compared based on different parameters such as stability and accuracy of prediction. The techniques are benchmarked using Fashion MNIST dataset**[1] **by training the model using 6000 training samples and testing the trained model using 1000 samples from the test dataset. Various optimization techniques such as Polyak's classical momentum**[2]**, RMSprop and Adam Optimization**[3] **are studied and its efficiency analyzed.**

***Index Terms*— Adam, Classification, Deep Neural Network, Momentum, Gradient Optimization, Polyak's classical momentum, RMSprop**

I. Introduction

THE field of Deep learning has received greater attention among research community in recent years. This could be attributed to the large scale application of this technology and the efficiency with which they can be used to develop simple solutions for a number of complex problems. The power of a neural network lies in its ability to learn from data efficiently in a short time and provide accurate prediction. The cost function of a Neural Network, which is defined as the difference between the desired output and the actual prediction generated by the network, should be minimal for an efficient Neural Network. A number of algorithms have been developed to minimize this cost function and one of the most successful methods is the Gradient Descent algorithm.

Gradient descent minimizes a mathematical objective function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient, i.e. an objective function *J(θ)* parameterized by a model's parameters $\theta \in \mathbb{R}^d$ is minimized by updating the parameters in the opposite direction of the gradient and a local minimum is then reached. Momentum in Gradient Descent is a technique which helps the objective function to converge faster by accelerating the gradient vectors in the optimal direction. While updating momentum, the magnitude of the gradient will be larger and therefore the learning rate needs to be smaller.

In this paper, a 3-layer deep neural network was implemented with the input layer containing 784 input neurons to receive the grayscale level from 784 pixels (28 * 28) and the first hidden layer contains 500 neurons which is connected to the second layer containing 100 neurons. The third or the output layer contains 10 neurons which corresponds to each of the ten categories in the Fashion MNIST dataset. We have used Rectified Linear Unit (ReLU) as activation function for hidden layers and the Softmax function (Normalized exponential function) as the activation function for output layer.

II. Related work

The momentum method in Gradient descent was proposed by Rumelhart, Hinton and Williams' seminal paper on backpropagation learning[4]. The gradient descent with momentum remembers the update Δw at each iteration and determines the next update as a linear combination of the gradient and the previous update[5].

$$\Delta w := \alpha \Delta w - \eta \nabla Q_i(w)$$
$$w := \Delta w + w$$

that leads to,

$$w := w - \eta \nabla Q_i(w) + \alpha \Delta w$$

where the parameter w which minimizes $Q_i(w)$ is to be estimated, and η is a step size (sometimes called the learning rate in machine learning).The main idea behind momentum is to accelerate progress along the dimension which the gradient points to and slowly progress along the dimensions where the sign of the gradient continues to change.[6]

This simple idea has been used for decades to provide optimal results in gradient descent. A brief description of the different gradient descent optimization techniques that have been used is as follows:

A. *Polyak's classical momentum*

Polyak (1964) showed that the aforementioned concept of momentum can considerably accelerate convergence to a local minimum, requiring $\sqrt{R}$ times fewer iterations than steepest descent to give the same level of accuracy, where R is the condition number of the curvature at the minimum and η is set to $(\sqrt{R} - 1)\ (\sqrt{R} + 1)$.

B. *RMSprop*

RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class.[7]

RProp: This combines the idea of using only the sign of the gradient with the idea of adapting the step size separately for each weight.

RProp would increment the weight nine times and decrement it once by about the same amount(So that the weight would grow a lot). RProp is equivalent to using the gradient but also dividing by the size of the gradient.

RMSProp: Keep a moving average of the squared gradient for each weight.

$MeanSquare(w,t) = 0.9*MeanSquare(w, t-1)+0.1(\partial E/\partial w^{(t)})^2$

Dividing the gradient by $\sqrt{MeanSquare(w,t)}$ makes the learning work much better.

C. Adams's optimization

Adaptive Momentum Estimation, as the name suggests, computes adaptive learning rates for each parameter. Similar to momentum, this technique keeps an exponentially decaying average of the past gradients m_t. On an error surface, the Adam optimization technique leads to a flat minima.

Let m_t and v_t be the estimates of the mean and uncentered variance of the gradients respectively.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

These estimates are biased towards zero as they are initialized as vectors of 0's. To resolve this, the bias corrected estimates are computed.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Updating the above parameters yields the Adam update rule as

$$\Theta_{t+1} = \theta_t - \eta \frac{\eta}{\sqrt{\hat{v}_t + \varepsilon}} \hat{m}_t$$

β_1 is initialized to a default value of 0.9, β_2 to a default value of 0.999 and ε to a default value of 10^{-8} in this optimization technique.

III. METHOD

The fashion MNIST dataset was used for analyzing the performance of different gradient descent techniques using mini-batch implementation.

A. Fashion MNIST

Fashion-MNIST is a dataset consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. The dataset contains ten categories of clothing and accessories.

B. Neural Network Configuration

A three-layer Neural Network was implemented. The First layer is the input layer which contains 784 neurons matching with the image size of the fashion MNIST dataset. There are two hidden layers, each containing 500 and 100 neurons respectively. The last layer contains 10 neurons matching with the number of categories that are to be classified.

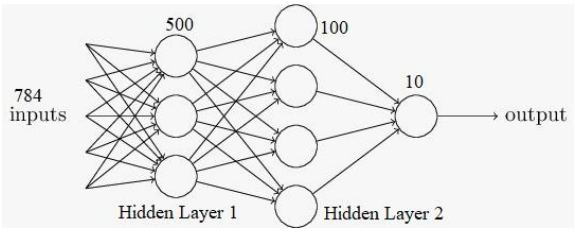

Fig.1 Neural Network Architecture

C. Activation Function

Rectified Linear Units(ReLU) is used as the activating function for all layers except the output layer. The softmax function is used for calculating the cross-entropy loss between the predicted and actual labels.

IV. EXPERIMENTS

A. Mini Batch

Mini-batches of size 10,15 and 20 were used to compare the performances of the different gradient descent techniques.

B. Learning Rate

The learning rate was varied between 0.00001 to 0.1 and the loss for each configuration was evaluated.

C. Epoch

The loss rate for 300, 500 and 700 epochs were studied.

V. INFERENCES

A. Choosing Learning Rate

Choosing the ideal learning rate is a challenge. If learning rate is lowered, the convergence is very slow. If the learning rate is high, it can hinder convergence and cause the loss function to fluctuate. Different models with different learning rates were evaluated and the results are given in the tables below:

TABLE I. GRADIENT DESCENT WITH NO MOMENTUM

Learning rate	Training Accuracy	Testing Accuracy
0.01	84.05	83.60
0.07	88.80	85.20
0.1	**89.18**	**85.30**

TABLE II. GRADIENT DESCENT WITH POLYAK'S MOMENTUM

Learning rate	Training Accuracy	Testing Accuracy
0.01	95.50	86.00
0.07	87.85	85.00
0.1	**96.83**	**86.0**

TABLE III. GRADIENT DESCENT WITH RMSPROP

Learning rate	Training Accuracy	Testing Accuracy
0.00001	77.50	74.50
0.00007	76.03	75.00
0.0007	**97.65**	**85.50**
0.0001	88.50	85.40

TABLE IV. GRADIENT DESCENT WITH ADAM

Learning rate	Training Accuracy	Testing Accuracy
0.00007	86.83	84.90
0.0007	**99.43**	**86.90**
0.0001	88.78	85.70

B. Analysis of Loss with training cycles

Initially, the training loss and validation loss are high, but the system quickly learns to predict the labels with higher accuracy as the number of training cycles increase. Also, the loss rate decreases rapidly which is represented by a steep curve in the plots given below, but it flattens out at the end as

it approaches convergence. Here, plots with different learning rates are compared to identify the parameters giving maximum efficiency.

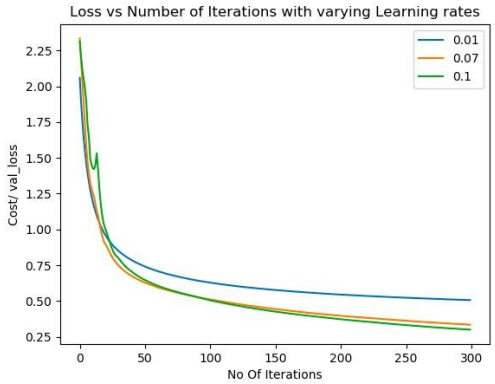

Fig. 2 Plot of Loss with training cycles (No Momentum)

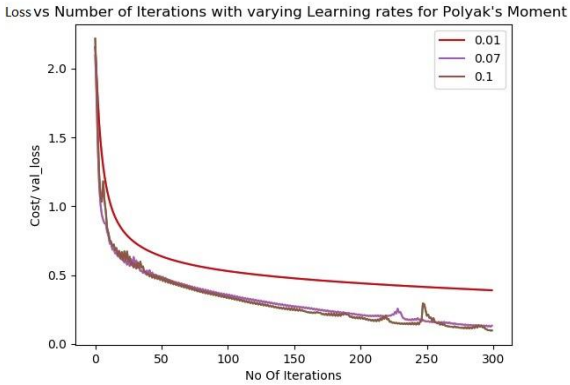

Fig. 3 Plot of Loss with training cycle (Polyak's Momentum)

C. *RMSProp and Adam*

RMSprop and Adam avoids the problem of radically diminishing gradients by storing an exponentially decaying average of past squared gradients and thus avoid fluctuations and recovers quickly from saddle point. Hence the plots shown in Figure 4 and 5 are much smoother with little noise associated with them. Additionally, these two techniques help in finding the right direction for descent and converges quickly compared to classical approach.

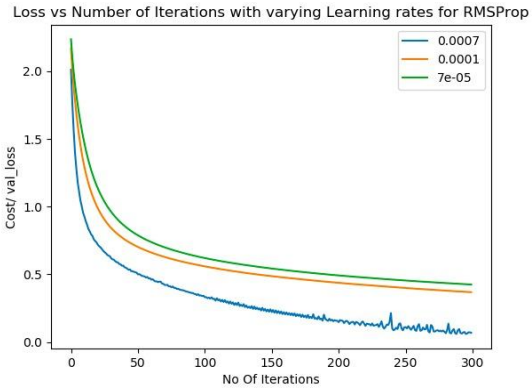

Fig. 4 Plot of Loss with training cycles (RMSprop)

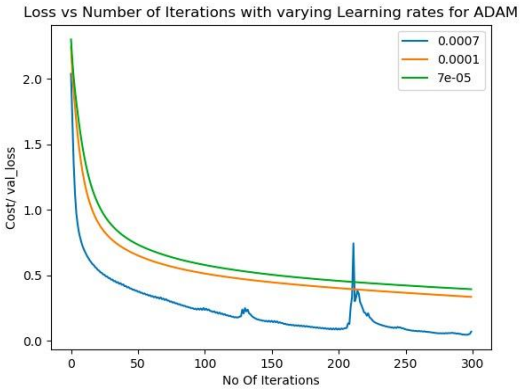

Fig. 5 Plot of Loss with training cycles (Adam)

D. *Batch Size:*

Varying the batch size affects the amount of fluctuations in the plot. When the batch sizes are small, the gradients are noisy and this reflects as spikes in the plot. Increasing the sizes of the batches to a certain threshold helps in reducing the fluctuations in the plot. Figures 6 and 7 shows the results of the varying batch sizes between 15 and 20.

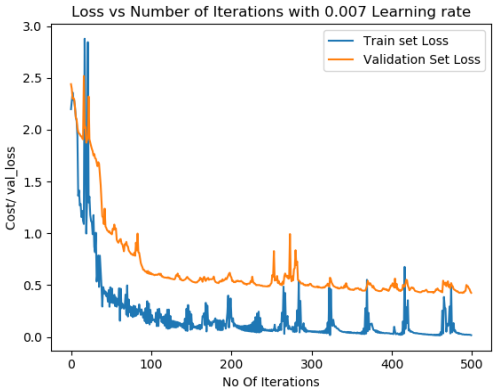

Fig. 6 Adam with batch size 15

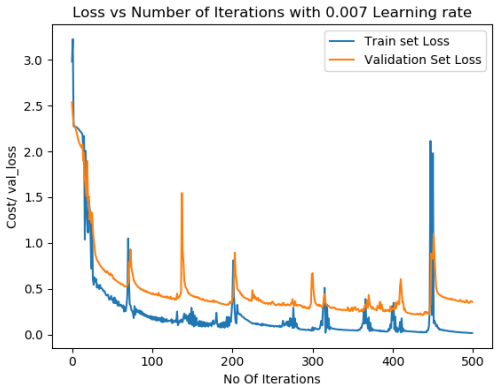

Fig. 7 Adam with batch size 20

E. Overall Performance

The performance of gradient descent algorithms were compared by changing the parameters such as number of epochs and batch size. The corresponding accuracies for each setting is shown in the tables below. It is inferred that the accuracy generally increases as the number of iterations and the batch size increases till a threshold.

TABLE.V. GRADIENT DESCENT WITH NO MOMENTUM AND LR=0.1

Number of Iterations	Batch size	Training accuracy	Testing accuracy
500	10	92.16	86.10
500	15	95.86	86.20
500	20	97.05	85.50
700	10	93.63	86.00
700	15	97.40	86.60
700	20	98.40	85.80

TABLE VI. GRADIENT DESCENT WITH POLYAK'S MOMENTUM AND LR=0.1

Number of Iterations	Batch size	Training accuracy	Testing accuracy
500	10	97.81	85.20
500	15	98.36	86.40
500	20	86.61	85.40
700	10	99.86	85.60
700	15	99.61	85.70
700	20	99.70	86.60

TABLE VII. GRADIENT DESCENT WITH RMSPROP AND LR=0.00007

Number of Iterations	Batch size	Training accuracy	Testing accuracy
500	10	89.45	86.20
500	15	90.45	84.80
500	20	88.43	82.90
700	10	91.21	86.20
700	15	92.03	84.70
700	20	89.30	81.50

TABLE VIII. GRADIENT DESCENT WITH ADAM AND LR=0.007

Batch Size	Epochs	Learning Rate	No Momentum	Nesterov	RMSProp	Adam
100	50	0.005	86.24%	86.29%	10%	88.78%
1000	100	0.00005	10%	22.88%	86.89%	87.14%
1000	100	0.5	88.99%	88.47%	10%	10%
100	100	0.00005	60.51%	61.25%	89.10%	88.95%
100	100	0.5	89.95%	88.95%	10%	10%

F. Different variants of Gradient techniques

Stochastic Gradient Descent (SGD), which updates the model parameters for every training data, provides an immediate insight into the performance and accuracy of the model at the cost of the number of computations. While Batch gradient is relatively faster and provides a smooth gradient descent because the updates are much fewer. Mini-batch finds a perfect balance between performance and robustness from both SGD and Batch gradient by splitting the data into several batches and providing better performance with relatively fewer updates.

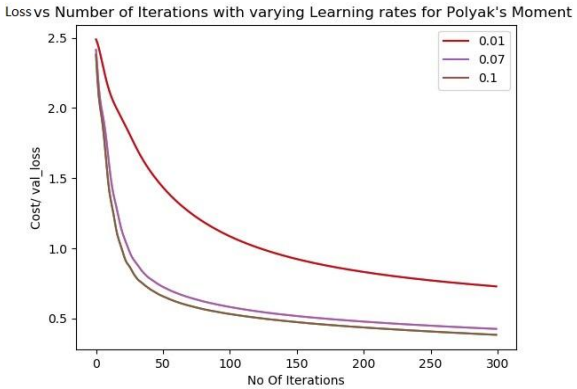

Fig. 8 Batch implementation For Polyak's Momentum

TABLE IX. COMPARISON WITH INBUILT LIBRARY FUNCTIONS

Epoch	LR	No Momentum	Nesterov	RMSProp	Adam
50	0.005	86.24%	86.29%	10%	88.78%
100	0.00005	10%	22.88%	86.89%	87.14%
100	0.5	88.99%	88.47%	10%	10%
100	0.00005	60.51%	61.25%	89.10%	88.95%
100	0.5	89.95%	88.95%	10%	10%

Table 9 shows the various evaluations performed using the inbuilt functions of the Keras library on the same dataset. The accuracies which we obtain are comparable to the inbuilt functions, but their running time was relatively quicker.

VI. Conclusion

Through several executions of training cycles, we evaluated the performances of each technique by varying Batch sizes, learning rates and number of epochs. RMSprop and ADAM showed best performances and accuracies at lower learning rates and the results were robust against fluctuations and noise introduced by the configurations.

VII. Division of work

Gradient Descent with no momentum, ADAM's Optimization	Arun Vignesh Malarkkan
RMSprop, Gradent Descent Optimization evaluation using Inbuilt functions	Balaji Gokulakrishnan
Performance Evaluation and Analysis	Gowtham Sekkilar
Polyak's Classical Momentum, Mini Batch implementation	Raghavendran Ramakrishnan

VIII. Self-Peer Evaluation Table

Arun Vignesh Malarkkan	20
Balaji Gokulakrishnan	20
Gowtham Sekkilar	20
Raghavendran Ramakrishnan	20

References

[1] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms," pp. 1–6, 2017.

[2] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Comput. Math. Math. Phys.*, vol. 4, no. 5, pp. 1–17, Jan. 1964.

[3] D. P. Kingma and J. Lei Ba, "ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION."

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, p. 533, Oct. 1986.

[5] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," 2013.

[6] M. D. Zeiler, "ADADELTA: AN ADAPTIVE LEARNING RATE METHOD."

[7] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," COURSERA: Neural networks for machine learning, vol. 4, no. 2, pp. 26–31, 2012.

CODE SNIPPETS

Gradient Descent Optimization with No Momentum:

```
def update_parameters(parameters, gradients, epoch, learning_rate, decay_rate=0.0):
    alpha = learning_rate*(1/(1+decay_rate*epoch))
    L = len(parameters)//2
    for i in range(L):
        parameters["W" + str(i+1)] -= alpha * gradients["dW" + str(i+1)]
        parameters["b" + str(i+1)] -= alpha * gradients["db" + str(i+1)]

    return parameters, alpha
```

Gradient Descent Optimization with Polyak's Momentum:

```
def update_parameters_with_momentum(parameters, gradients, epoch, v, beta, learning_rate,
decay_rate=0.01):

    alpha = learning_rate*(1/(1+decay_rate*epoch))
    L = len(parameters) // 2 # number of layers in the neural networks

    # Momentum update for each parameter
    for i in range(L):

        # compute velocities
        v["dW" + str(i + 1)] = beta * v["dW" + str(i + 1)] + (1 - beta) * gradients['dW' +
str(i + 1)]
        v["db" + str(i + 1)] = beta * v["db" + str(i + 1)] + (1 - beta) * gradients['db' +
str(i + 1)]
        # update parameters
        parameters["W" + str(i + 1)] = parameters["W" + str(i + 1)] - learning_rate * v["dW" +
str(i + 1)]
        parameters["b" + str(i + 1)] = parameters["b" + str(i + 1)] - learning_rate * v["db" +
str(i + 1)]


    return parameters, alpha, v
```

Gradient Descent Optimization with RMSprop:

```
def update_parameters_with_adam(parameters, gradients, epoch, v, s, t, learning_rate=0.007,
beta1=0.9, beta2=0.999, epsilon=1e-8, decay_rate = 0.01):

    L = len(parameters) // 2                 # number of layers in the neural networks
    v_corrected = {}                         # Initializing first moment estimate, python
dictionary
    s_corrected = {}                         # Initializing second moment estimate, python
dictionary
    alpha = learning_rate*(1/(1+decay_rate*epoch))

    # Perform RMSprop update on all parameters
    for l in range(L):
        # Computing 'G' for RMSProp.
```

```
        s["dW" + str(l + 1)] = beta1 * s["dW" + str(l + 1)] + (1 - beta1) *
np.power(gradients['dW' + str(l + 1)], 2)
        s["db" + str(l + 1)] = beta1 * s["db" + str(l + 1)] + (1 - beta1) *
np.power(gradients['db' + str(l + 1)], 2)

        # Update parameters. Inputs: "parameters, learning_rate, v_corrected, s_corrected,
epsilon". Output: "parameters".

        parameters["W" + str(l + 1)] = parameters["W" + str(l + 1)] - learning_rate *
gradients['dW' + str(l + 1)]  / np.sqrt(s["dW" + str(l + 1)] + epsilon)
        parameters["b" + str(l + 1)] = parameters["b" + str(l + 1)] - learning_rate *
gradients['db' + str(l + 1)]  / np.sqrt(s["db" + str(l + 1)] + epsilon)

return parameters, alpha, v, s
```

Gradient Descent Optimization with Adam:

```
def update_parameters_with_adam(parameters, gradients, epoch, v, s, t, learning_rate=0.007,
beta1=0.9, beta2=0.999, epsilon=1e-8, decay_rate = 0.01):

    L = len(parameters) // 2                 # number of layers in the neural networks
    v_corrected = {}                         # Initializing first moment estimate, python
dictionary
    s_corrected = {}                         # Initializing second moment estimate, python
dictionary
    alpha = learning_rate*(1/(1+decay_rate*epoch))

    # Perform Adam update on all parameters
    for l in range(L):
        # Moving average of the gradients. Inputs: "v, grads, beta1". Output: "v".
        v["dW" + str(l + 1)] = beta1 * v["dW" + str(l + 1)] + (1 - beta1) * gradients['dW' +
str(l + 1)]
        v["db" + str(l + 1)] = beta1 * v["db" + str(l + 1)] + (1 - beta1) * gradients['db' +
str(l + 1)]

        # Compute bias-corrected first moment estimate. Inputs: "v, beta1, t". Output:
"v_corrected".
        v_corrected["dW" + str(l + 1)] = v["dW" + str(l + 1)] / (1 - np.power(beta1, t))
        v_corrected["db" + str(l + 1)] = v["db" + str(l + 1)] / (1 - np.power(beta1, t))

        # Moving average of the squared gradients. Inputs: "s, grads, beta2". Output: "s".
        s["dW" + str(l + 1)] = beta2 * s["dW" + str(l + 1)] + (1 - beta2) *
np.power(gradients['dW' + str(l + 1)], 2)
        s["db" + str(l + 1)] = beta2 * s["db" + str(l + 1)] + (1 - beta2) *
np.power(gradients['db' + str(l + 1)], 2)

        # Compute bias-corrected second raw moment estimate. Inputs: "s, beta2, t". Output:
"s_corrected".
        s_corrected["dW" + str(l + 1)] = s["dW" + str(l + 1)] / (1 - np.power(beta2, t))
        s_corrected["db" + str(l + 1)] = s["db" + str(l + 1)] / (1 - np.power(beta2, t))

        # Update parameters. Inputs: "parameters, learning_rate, v_corrected, s_corrected,
epsilon". Output: "parameters".

        parameters["W" + str(l + 1)] = parameters["W" + str(l + 1)] - learning_rate *
v_corrected["dW" + str(l + 1)] / np.sqrt(s_corrected["dW" + str(l + 1)] + epsilon)
        parameters["b" + str(l + 1)] = parameters["b" + str(l + 1)] - learning_rate *
v_corrected["db" + str(l + 1)] / np.sqrt(s_corrected["db" + str(l + 1)] + epsilon)

    return parameters, alpha, v, s
```