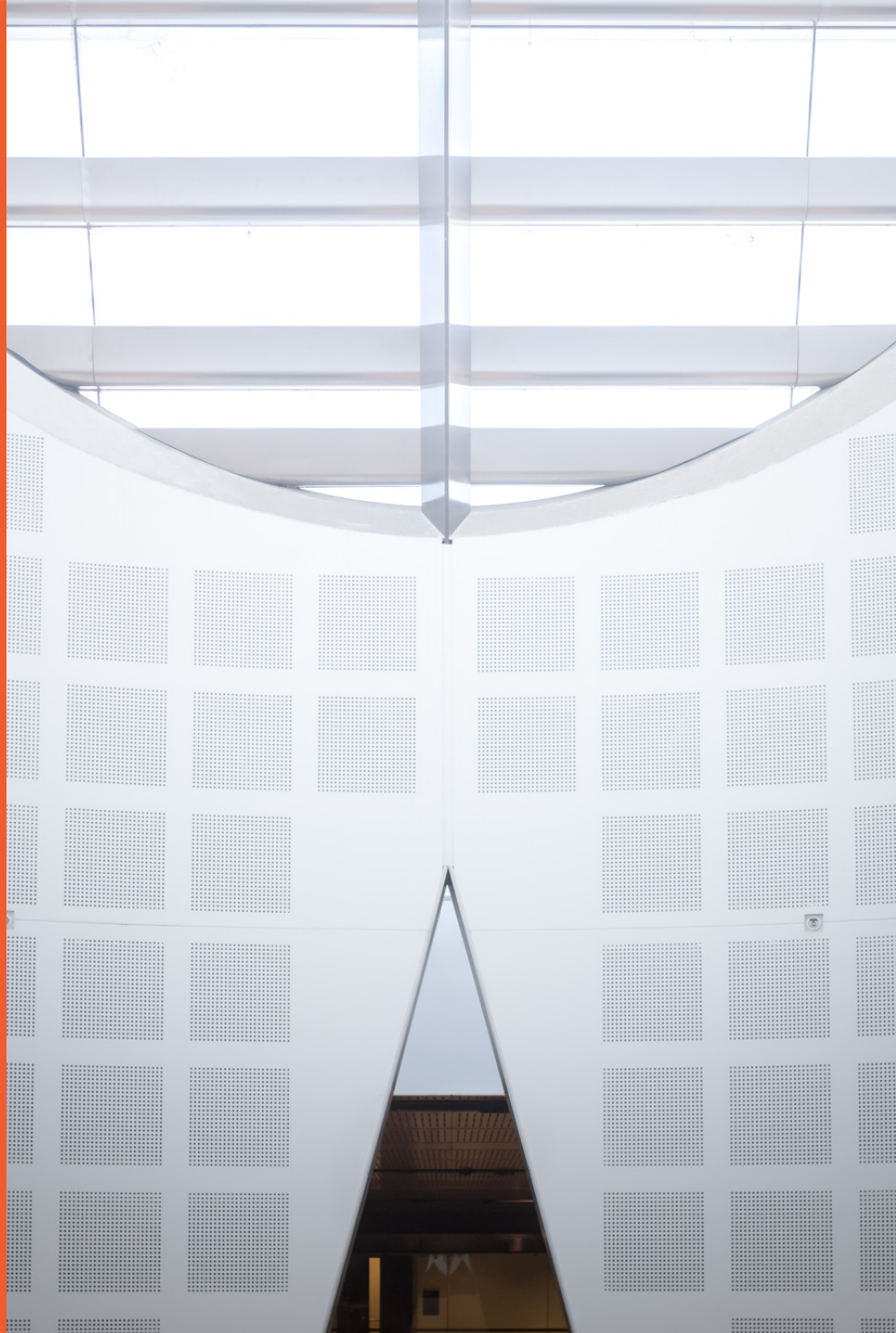
Optimization for Training Deep Models

Dr Chang Xu

School of Computer Science

THE UNIVERSITY OF SYDNEY



A common training process for neural networks

1. Initialize the parameters

2. Choose an optimization algorithm

3. Repeat these steps:
    1. Forward propagate an input
    2. Compute the cost function
    3. Compute the gradients of the cost with respect to parameters using backpropagation
    4. Update each parameter using the gradients, according to the optimization algorithm

Outline

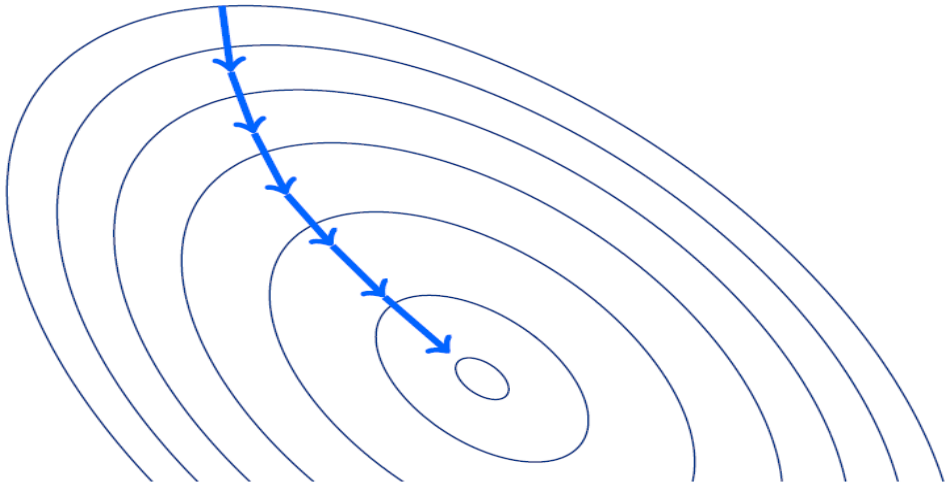
1. Gradient Descent Optimization
   - Gradient descent
   - Momentum
   - Nesterov
   - Adagrad
   - Adadelta
   - Rmsprop
   - Adam
   - Adamax
2. Initialization

Gradient descent

- $J(\theta, \mathcal{X})$ is the objective function, θ are the parameters, $\mathcal{X}$ is the training dataset. We want to optimize $J(\theta, \mathcal{X})$ by:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta, \mathcal{X})$$

where $\nabla_\theta J(\theta, \mathcal{X})$ is the gradient of objective function w.r.t. the parameters. η is the learning rate.



Gradient descent

- **Batch gradient descent:**

  Compute the gradient for the **entire** training dataset.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta, \mathcal{X}^{(1:end)})$$

- **Stochastic gradient descent:**

  Compute the gradient for **each** training example.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta, \mathcal{X}^{(i)})$$

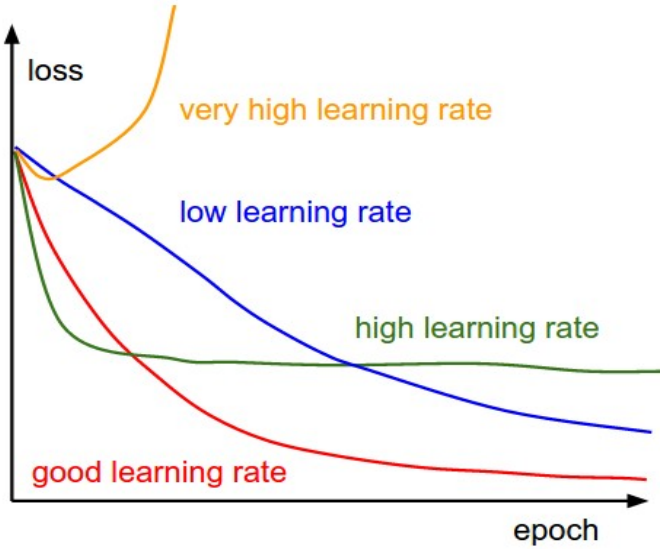
- **Mini-batch gradient descent:**

  Compute the gradient for every **mini-batch** training examples.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta, \mathcal{X}^{(i:i+n)})$$

Some Challenges For Gradient descent

- **Choosing a proper learning rate can be difficult.**
  - A learning rate that is too small leads to painfully slow convergence.
  - A learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.



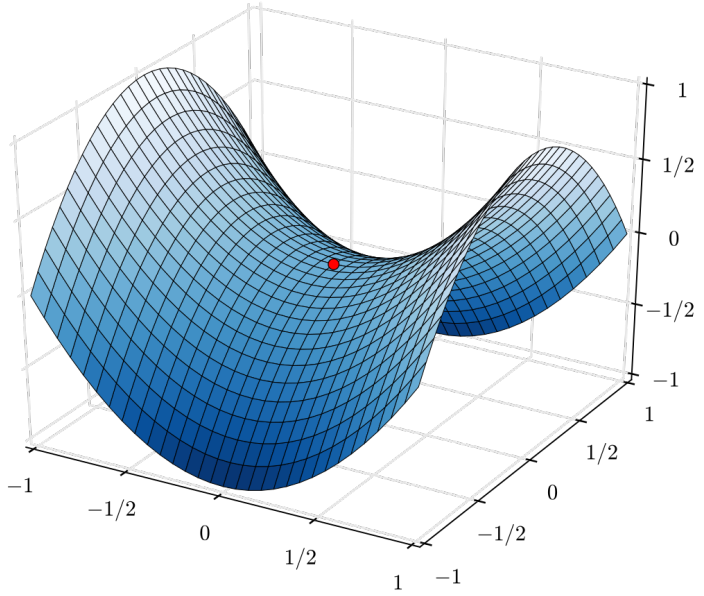
http://srdas.github.io/DLBook/GradientDescentTechniques.html

Some Challenges For Gradient descent

- **The same learning rate applies to all parameter updates.**
  - If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.

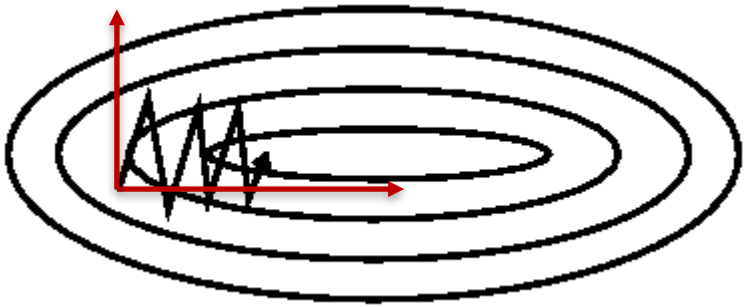
Some Challenges For Gradient descent

- **Easily get trapped in numerous saddle points.**
  - Saddle points are points where one dimension slopes up and another slopes down. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

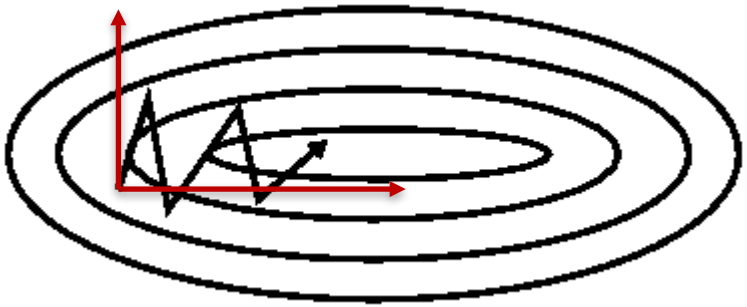


Momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima.



Without Momentum



With Momentum

https://www.willamette.edu/~gorr/classes/cs449/momrate.html

Momentum tries to accelerate SGD in the relevant direction and dampens oscillations.

Momentum

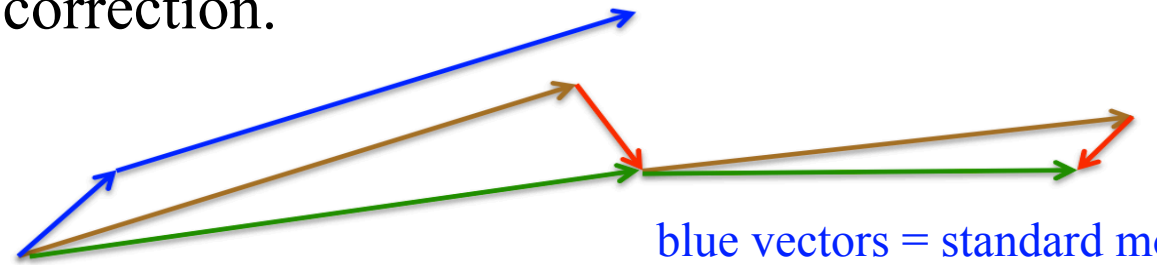
The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta_t = \theta_{t-1} - v_t$$

Momentum term γ is usually set to 0.9 or a similar value.

Nesterov accelerated gradient (NAG)

- The **standard momentum** method:
  - First computes the gradient at the current location
  - Then takes a big jump in the direction of the updated accumulated gradient.
- **Nesterov accelerated gradient**（often works better）.
  - First make a big jump in the direction of the previous accumulated gradient.
  - Then measure the gradient where you end up and make a correction.



https://isaacchanghau.github.io/2017/05/29/parameter-update-methods/

blue vectors = standard momentum
brown vector = jump, red vector = correction,
green vector = accumulated gradient ,

Nesterov accelerated gradient (NAG)

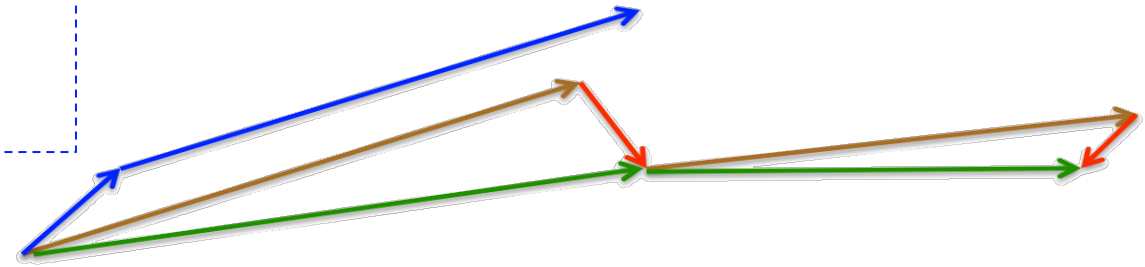
First make a big jump in the direction of the previous accumulated gradient.
Then measure the gradient where you end up.
Finally accumulate the gradient and make a correction.

First make a big jump

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Then measure the gradient where you end up

Finally accumulate the gradient and make a correction



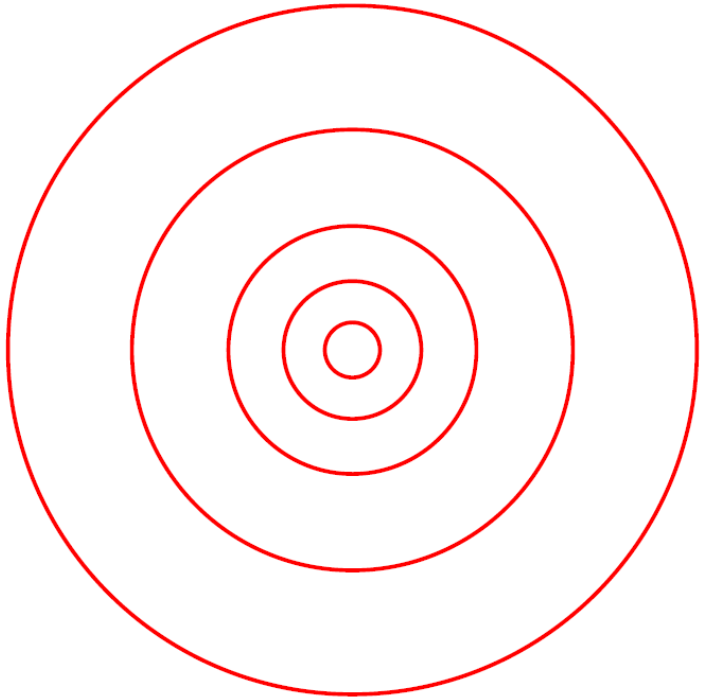
blue vectors = standard momentum
brown vector = jump, red vector = correction,
green vector = accumulated gradient ,

Adaptive Learning Rate Methods

Motivation

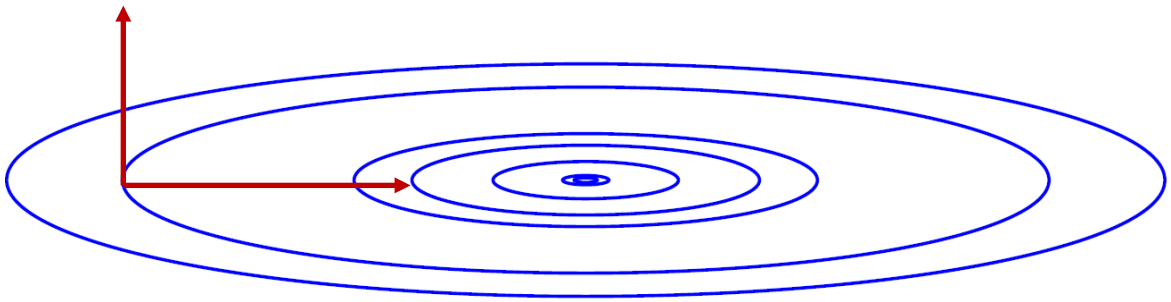
- Till now we assign the same learning rate to all features.
- If the feature vary in importance and frequency, why is this a good idea?
- It’s probably not!

Motivation



Nice (all features are equally important)

Motivation



Harder

- **Adagrad** $\theta_t = [\theta_{t,1}, \theta_{t,2}, \cdots, \theta_{t,i}, \cdots, \theta_{t,d}]$

- **Original gradient descent:** The same for all parameters

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot \nabla_\theta J(\theta_i)$$

Perform an update for all parameters using the same learning rate.

- **Adagrad:** Adapt the learning rate to the parameters based on the past gradients that have been computed for θ_i.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot \nabla_\theta J(\theta_i)$$

$G_t \in R^{d \times d}$ is a diagonal matrix where each diagonal element is the sum of the squares of the gradients w.r.t. θ_i up to time step t.

Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

where

$$g_{t,i} = \left(\nabla_\theta J(\theta_t)\right)_i \qquad G_{t,ii} = \sum_{i=1}^{t} g_{t,i}^2$$

- **Adapt the learning rate to the parameters.**
  - Perform larger updates for infrequent updated parameters, and smaller updates for frequent updated parameters.
- **Well-suited for dealing with sparse data.**

Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

where

$$g_{t,i} = \left(\nabla_\theta J(\theta_t)\right)_i \qquad G_{t,ii} = \sum_{i=1}^{t} g_{t,i}^2$$

- **The learning rate eventually becomes infinitesimally small.**
  - $G_{t,ii} = \sum_{i=1}^{t} g_{t,i}^2$ increases monotonically, $\frac{\eta}{\sqrt{G_{t,ii}+\epsilon}}$ will eventually be infinitesimally small.
- **Need to manually select a global learning rate .**

Adadelta

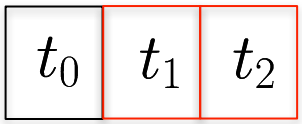
- In Adagrad, the denominator accumulates the squared gradients from each iteration.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

where

$$g_{t,i} = \left(\nabla_\theta J(\theta_t)\right)_i \qquad G_{t,ii} = \sum_{i=1}^{t} g_{t,i}^2$$

➢ **Accumulate over window.**



If we set window size to 2, we can get that

$$t = \frac{1}{2}(t_1 + t_2)$$

Adadelta - Accumulate over window

- Storing fixed previous squared gradients cannot accumulate to infinity and instead becomes a local estimate using recent gradients.
- Adadelta implements it as an exponentially decaying average of all past squared gradients.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$RMS[g]_t = \sqrt{E[g^2]_t + \epsilon}$$

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

Adadelta

- Need for a manually selected global learning rate.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

where

$$g_{t,i} = \left(\nabla_\theta J(\theta_t)\right)_i \qquad G_{t,ii} = \sum_{i=1}^{t} g_{t,i}^2$$

➢ **Correct units with Hessian approximation.**

Adadelta - Correct units with Hessian approximation

- The units in SGD and Momentum relate to the gradient, not the parameter.

$$\theta = \theta - \eta \cdot g$$

$$units\ of\ \Delta\theta \propto units\ of\ g \propto \frac{\partial J}{\partial \theta} \propto \frac{1}{units\ of\ \theta}$$

- Second order methods such as Newton's method do have the correct units.

$$\theta = \theta - H^{-1} g$$

$$units\ of\ \Delta\theta \propto H^{-1} g \propto \frac{\frac{\partial J}{\partial \theta}}{\frac{\partial^2 J}{\partial \theta^2}} \propto units\ of\ \theta$$

* Assume cost function J is unitless.

Adadelta - Correct units with Hessian approximation

$$H^{-1}g = \Delta\theta$$

$$\Delta\theta = \frac{\frac{\partial J}{\partial \theta}}{\frac{\partial^2 J}{\partial \theta^2}} \Rightarrow H^{-1} \propto \frac{1}{\frac{\partial^2 J}{\partial \theta^2}} \propto \frac{\frac{\partial J}{\partial \theta}}{} \propto \frac{\Delta\theta}{g}$$

We assume the curvature is locally smooth and approximate $\Delta\theta_t$ by computing the exponentially decaying RMS of $\Delta\theta$.

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t \longrightarrow \Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

RMSprop

- RMSprop and Adadelta have both been developed independently around the same time.
- RMSprop in fact is identical to the first update vector of Adadelta.

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

Adam

Combine the advantages of **Adgrad** and **RMSprop**.

- Adgrad: adaptive learning rate for different parameters.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} m_t$$

- RMSprop: exponentially decaying average.
  - Store an exponentially decaying average of past squared gradients v_t .

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

Keep an exponentially decaying average of past gradients m_t

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

Adam - bias-correct

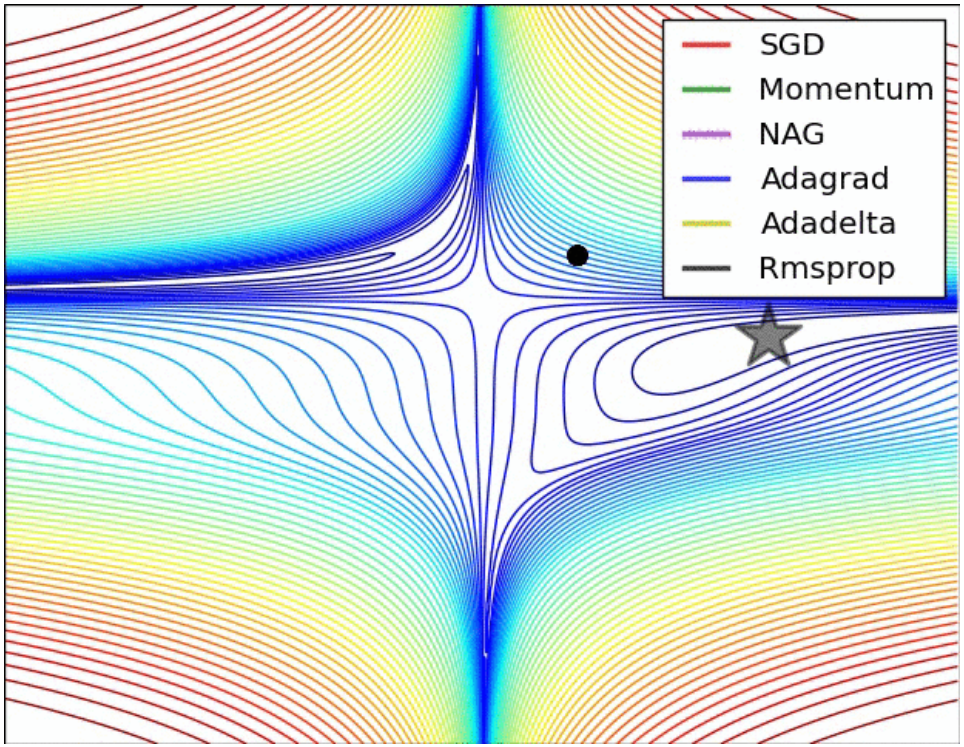
Since m_t and v_t are initialized as vectors of 0's, they are biased towards zero, especially during the initial time steps or when the decay rates are small (i.e. close to 1).

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

$$\begin{cases} \hat{m}_t = \dfrac{m_t}{1-\beta_1^t} \\ \hat{v}_t = \dfrac{v_t}{1-\beta_2^t} \end{cases} \longrightarrow \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

Visualization

- Both run towards the negative gradient direction
- Momentum and NAG run faster in the relevant direction



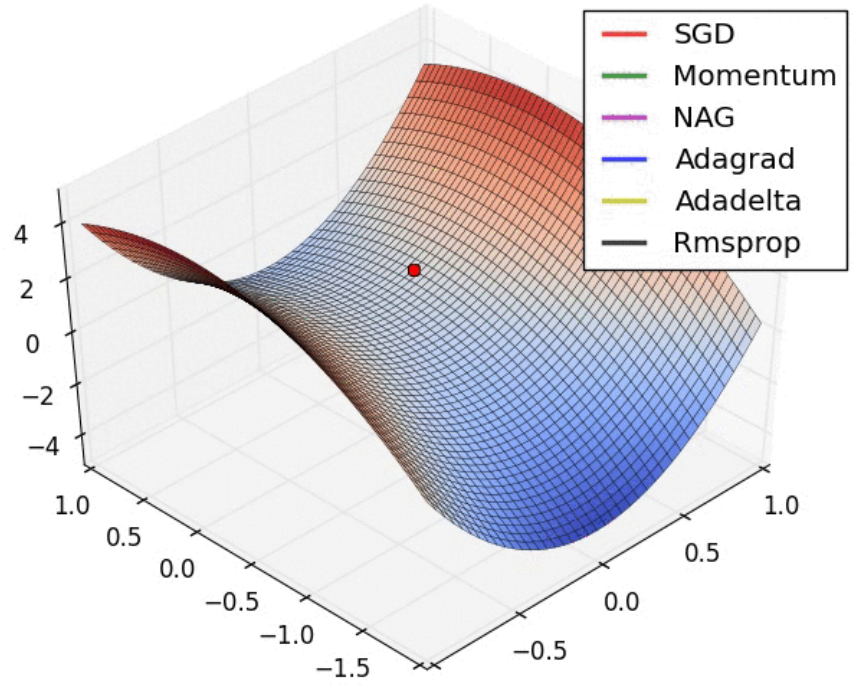
Contours of a loss face

http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Visualization

Saddle point: a point where one dimension has a positive slope, while the other dimensions has negative slopes.

- It's easier for **Rmsprop**, **Adadelta**, and **Adagrad** to escape the saddle point.



Behaviour around a saddle point

http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

1. Gradient Descent Optimization

Methods	Adaptive lnearning rate	Consistent units	Easily hand Saddle points
SGD	No	No	No
Momentum	No	No	No
Nesterov	No	No	No
Adagrad	Yes	No	Yes
Adadelta	Yes	Yes	Yes
Rmsprop	Yes	No	Yes
Adam	Yes	No	Yes

A common training process for neural networks

1. Initialize the parameters

2. Choose an optimization algorithm

3. Repeat these steps:
    1. Forward propagate an input
    2. Compute the cost function
    3. Compute the gradients of the cost with respect to parameters using backpropagation
    4. Update each parameter using the gradients, according to the optimization algorithm

Outline

1. Gradient Descent Optimization
   - Gradient descent
   - Momentum
   - Nesterov
   - Adagrad
   - Adadelta
   - Rmsprop
   - Adam
   - Adamax
2. **Initialization**

The initialization step can be critical to the model's ultimate performance, and it requires the right method.

Initialization

- All zero initialization
- Small random numbers
- Calibrating the variances

Initialization

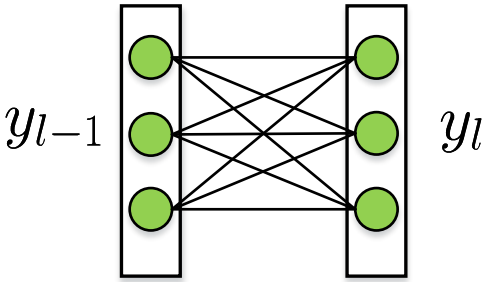
All zero initialization:

Every neuron computes the same output.

They will also compute the same gradients.

They will undergo the exact same parameter updates.

There will be no difference between all the neurons.



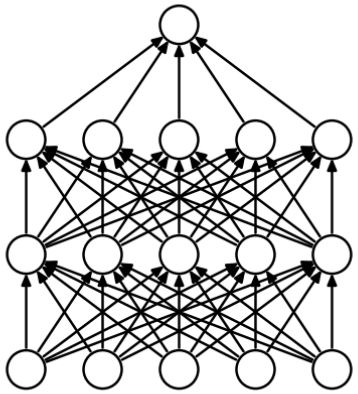
$$y_l = W y_{l-1} + b, \quad \frac{\partial \mathcal{L}}{\partial y_{l-1}} = W^\top \frac{\partial \mathcal{L}}{\partial y_l}$$
$$if \quad W = 0, \quad then \quad \frac{\partial \mathcal{L}}{\partial y_{l-1}} = 0$$
$$then, \quad \frac{\partial \mathcal{L}}{\partial y_m} = 0, \quad m = 0, ...l-1.$$

Initialize weights with values too small or too large

Despite breaking the symmetry, initializing the weights with values (i) too small or (ii) too large leads respectively to (i) slow learning or (ii) divergence.

Assuming all the activation functions are linear (identity function), we have

$$\hat{y} = W^{[L]}W^{[L-1]}W^{[L-2]} \ldots W^{[3]}W^{[2]}W^{[1]}x$$



Case 1: A too-large initialization leads to exploding gradients

Case 2: A too-small initialization leads to vanishing gradients

```python
[docs]class Linear(Module):
    r"""Applies a linear transformation to the incoming data: :math:`y = xA^T

    Args:
        in_features: size of each input sample
        out_features: size of each output sample
        bias: If set to ``False``, the layer will not learn an additive bias.
            Default: ``True``

    Shape:
        - Input: :math:`(N, *, H_{in})` where :math:`*` means any number of
          additional dimensions and :math:`H_{in} = \text{in\_features}`
        - Output: :math:`(N, *, H_{out})` where all but the last dimension
          are the same shape as the input and :math:`H_{out} = \text{out\_fea

    Attributes:
        weight: the learnable weights of the module of shape
            :math:`(\text{out\_features}, \text{in\_features})`. The values a
            initialized from :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})`, where
            :math:`k = \frac{1}{\text{in\_features}}`
        bias:   the learnable bias of the module of shape :math:`(\text{out\_
                If :attr:`bias` is ``True``, the values are initialized from
                :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
                :math:`k = \frac{1}{\text{in\_features}}`

    Examples::
        >>> m = nn.Linear(20, 30)
        >>> input = torch.randn(128, 20)
        >>> output = m(input)
        >>> print(output.size())
        torch.Size([128, 30])
    """
    __constants__ = ['bias', 'in_features', 'out_features']

    def __init__(self, in_features, out_features, bias=True):
        super(Linear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(torch.Tensor(out_features, in_features))
        if bias:
            self.bias = Parameter(torch.Tensor(out_features))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self):
        init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        if self.bias is not None:
            fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in)
            init.uniform_(self.bias, -bound, bound)
```

https://pytorch.org/docs/stable/_modules/torch/nn/modules/linear.html#Linear

```python
class _ConvNd(Module):

    __constants__ = ['stride', 'padding', 'dilation', 'groups', 'bias',
                     'padding_mode', 'output_padding', 'in_channels',
                     'out_channels', 'kernel_size']

    def __init__(self, in_channels, out_channels, kernel_size, stride,
                 padding, dilation, transposed, output_padding,
                 groups, bias, padding_mode):
        super(_ConvNd, self).__init__()
        if in_channels % groups != 0:
            raise ValueError('in_channels must be divisible by groups')
        if out_channels % groups != 0:
            raise ValueError('out_channels must be divisible by groups')
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding
        self.dilation = dilation
        self.transposed = transposed
        self.output_padding = output_padding
        self.groups = groups
        self.padding_mode = padding_mode
        if transposed:
            self.weight = Parameter(torch.Tensor(
                in_channels, out_channels // groups, *kernel_size))
        else:
            self.weight = Parameter(torch.Tensor(
                out_channels, in_channels // groups, *kernel_size))
        if bias:
            self.bias = Parameter(torch.Tensor(out_channels))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self):
        init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        if self.bias is not None:
            fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in)
            init.uniform_(self.bias, -bound, bound)
```

`torch.nn.init.kaiming_uniform_(tensor, a=0, mode='fan_in', nonlinearity='leaky_relu')` [SOURCE]

Fills the input *Tensor* with values according to the method described in *Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification* - He, K. et al. (2015), using a uniform distribution. The resulting tensor will have values sampled from $\mathcal{U}(-\text{bound}, \text{bound})$ where

$$\text{bound} = \text{gain} \times \sqrt{\frac{3}{\text{fan\_mode}}}$$

$$\mathcal{U}\left(-\sqrt{\frac{6}{n}}, \sqrt{\frac{6}{n}}\right)$$

Also known as He initialization.

Parameters

- **tensor** – an n-dimensional *torch.Tensor*
- **a** – the negative slope of the rectifier used after this layer (only
- **with 'leaky_relu')** (*used*) –
- **mode** – either `'fan_in'` (default) or `'fan_out'`. Choosing `'fan_in'` preserves the mag variance of the weights in the forward pass. Choosing `'fan_out'` preserves the magnitude backwards pass.
- **nonlinearity** – the non-linear function (*nn.functional* name), recommended to use only wit `'leaky_relu'` (default).

Examples

```
>>> w = torch.empty(3, 5)
>>> nn.init.kaiming_uniform_(w, mode='fan_in', nonlinearity='relu')
```

`torch.nn.init.calculate_gain(nonlinearity, param=None)`

Return the recommended gain value for the given nonlinearity function. The values

nonlinearity	gain
Linear / Identity	1
Conv{1,2,3}D	1
Sigmoid	1
Tanh	$\frac{5}{3}$
ReLU	$\sqrt{2}$
Leaky Relu	$\sqrt{\frac{2}{1+\text{negative\_slope}^2}}$

https://pytorch.org/docs/stable/nn.init.html?highlight=kaiming_uniform#torch.nn.init.kaiming_uniform_

How to find appropriate initialization values

1. The *mean* of the activations should be zero.
2. The *variance* of the activations should stay the same across every layer.

Under these two assumptions, the backpropagated gradient signal should not be multiplied by values too small or too large in any layer. It should travel to the input layer without exploding or vanishing.

Initialization (Xavier) -- [Xavier Glorot, Yoshua.Bengio 2010]

How Xavier Initialization keeps the variance the same across every layer?

Assume the activation function is *tanh*. The forward propagation is:

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = \tanh(z^{[l]})$$

How we should initialize our weights such that $Var\left(a^{[l-1]}\right) = Var(a^{[l]})$?

Early on in the training, we are in the ***linear regime* of *tanh***. Values are small enough and thus $\tanh\left(z^{[l]}\right) \approx z^{[l]}$, meaning that

$$Var\left(a^{[l]}\right) = Var(z^{[l]})$$

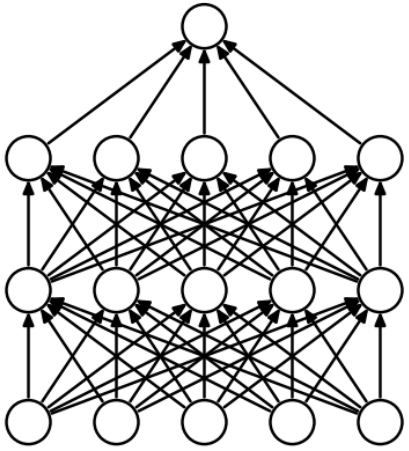
Initialization (Xavier) -- [Xavier Glorot, Yoshua.Bengio 2010]

How Xavier Initialization keeps the variance the same across every layer?

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = \tanh(z^{[l]})$$

$$Var\left(a^{[l]}\right) = Var\left(z^{[l]}\right)$$



$$z_k^{[l]} = \sum_{j=1}^{n^{[l-1]}} w_{kj}^{[l]} a_j^{[l-1]}$$

$$Var\left(a_k^{[l]}\right) = Var\left(z_k^{[l]}\right)$$

$$= Var\left(\sum_{j=1}^{n^{[l-1]}} w_{kj}^{[l]} a_j^{[l-1]}\right)$$

$$= \sum_{j=1}^{n^{[l-1]}} Var(w_{kj}^{[l]} a_j^{[l-1]})$$

1. Weights are independent and identically distributed
2. Inputs are independent and identically distributed
3. Weights and inputs are mutually independent

Initialization (Xavier) -- [Xavier Glorot, Yoshua.Bengio 2010]

How Xavier Initialization keeps the variance the same across every layer?

$$z_k^{[l]} = \sum_{j=1}^{n^{[l-1]}} w_{kj}^{[l]} a_j^{[l-1]}$$

$$Var(XY) = E[X]^2 Var(Y) + Var(X)E[Y]^2 + Var(X)Var(Y)$$

$$Var\left(a_k^{[l]}\right) = Var\left(z_k^{[l]}\right)$$

$$= Var\left(\sum_{j=1}^{n^{[l-1]}} w_{kj}^{[l]} a_j^{[l-1]}\right)$$

$$= \sum_{j=1}^{n^{[l-1]}} Var(w_{kj}^{[l]} a_j^{[l-1]})$$

$$Var\left(w_{kj}^{[l]} a_j^{[l-1]}\right)$$
$$= E\left[w_{kj}^{[l]}\right]^2 Var\left(a_j^{[l-1]}\right) + Var\left(w_{kj}^{[l]}\right) E\left[a_j^{[l-1]}\right]^2$$
$$+ Var\left(w_{kj}^{[l]}\right) Var(a_j^{[l-1]})$$

Weights are initialized with zero mean, and inputs are normalized.

$$Var\left(a_k^{[l]}\right) = n^{[l-1]} Var\left(w_{kj}^{[l]}\right) Var(a_j^{[l-1]})$$

Initialization (Xavier) -- [Xavier Glorot, Yoshua.Bengio 2010]

How Xavier Initialization keeps the variance the same across every layer?

$$Var\left(a_k^{[l]}\right) = n^{[l-1]} Var\left(w_{kj}^{[l]}\right) Var(a_j^{[l-1]})$$

$$Var\left(W^{[l]}\right) = \frac{1}{n^{[l-1]}}$$

This expression holds for every layer of our network:

$$Var\left(a^{[L]}\right) = n^{[L-1]} Var\left(W^{[L]}\right) Var\left(a^{[L-1]}\right)$$
$$= n^{[L-1]} Var\left(W^{[L]}\right) n^{[L-2]} Var\left(W^{[L-1]}\right) Var\left(a^{[L-2]}\right)$$
$$= \cdots = \left[\prod_{l=1}^{L} n^{[l-1]} Var\left(W^{[l]}\right)\right] Var(x)$$

Initialization (Xavier) -- [Xavier Glorot, Yoshua.Bengio 2010]

$$Var\left(a^{[L]}\right) = \left[\prod_{l=1}^{L} n^{[l-1]} Var\left(W^{[l]}\right)\right] Var(x)$$

$$n^{[l-1]} Var\left(W^{[l]}\right) \begin{cases} < 1 & \rightarrow \text{ Vanishing Signal} \\ = 1 & \rightarrow Var\left(a^{[L]}\right) = Var(x) \\ > 1 & \rightarrow \text{ Exploding Signal} \end{cases}$$

$$Var\left(W^{[l]}\right) = \frac{1}{n^{[l-1]}}$$

https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.xavier_normal_

Initialization (Xavier) -- [Xavier Glorot, Yoshua.Bengio 2010]

We worked on activations computed during the forward propagation, and get

$$Var(W^{[l]}) = \frac{1}{n^{[l-1]}}$$

The same result can be derived for the backpropagated gradients:

$$Var(W^{[l]}) = \frac{1}{n^{[l]}}$$

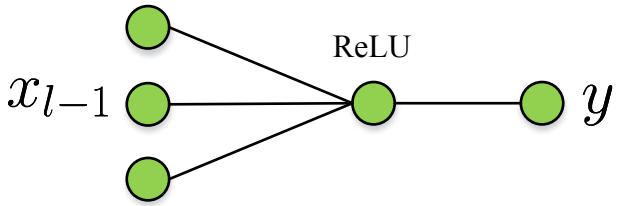
Xavier initialization would either initialize the weights as

xavier_normal $\quad \mathcal{N}(0, \frac{2}{n^{[l-1]} + n^{[l]}})$

xavier_uniform $\quad \mathcal{U}(-\frac{\sqrt{6}}{\sqrt{n^{[l-1]}+n^{[l]}}}, \frac{\sqrt{6}}{\sqrt{n^{[l-1]}+n^{[l]}}})$

Initialization（He） -- [Kaiming He, Xiangyu Zhang, et al. 2015]

If we use ReLU as the activation function, n is the number of inputs, then we have:

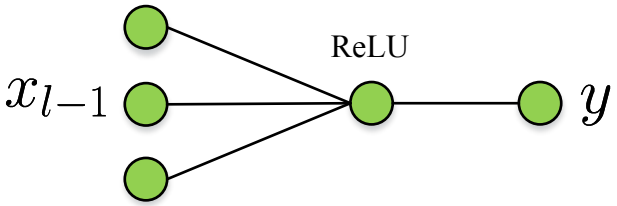


$$
\begin{aligned}
\mathrm{Var}(y) &= \mathrm{Var}(\sum_i^n w_i x_i) \\
&= \sum_i^n \mathrm{Var}(w_i x_i) \\
&= \sum_i^n [E(w_i)]^2 \mathrm{Var}(x_i) + [E(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= \sum_i^n [E(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= \sum_i^n [E(x_i)]^2 \mathrm{Var}(w_i) + (E[x_i^2] - [E(x_i)]^2)\mathrm{Var}(w_i) \\
&= \sum_i^n E[x_i^2]\mathrm{Var}(w_i) = (n\mathrm{Var}(w))E[x_i^2]
\end{aligned}
$$

Initialization（He） -- [Kaiming He, Xiangyu Zhang, et al. 2015]

If we use ReLU as the activation function, n is the number of inputs, then for layer l we have:

$$\mathrm{Var}(y_l) = (n\mathrm{Var}(w))\, E[x_{l-1}^2]$$
$$x_{l-1} = max(0, y_{l-1})$$



If y_{l-1} has zero mean, and has a symmetric distribution around zero:

$$E[x_{l-1}^2] = E[(max(0, y_{l-1}))^2] = \frac{1}{2}\mathrm{Var}[y_{l-1}]$$

So we have: $\mathrm{Var}(w) = \frac{2}{n}$

kaiming_normal $\mathcal{N}(0, \frac{2}{n})$

kaiming_uniform $\mathcal{U}(-\sqrt{\frac{6}{n}}, \sqrt{\frac{6}{n}})$

Thank you!