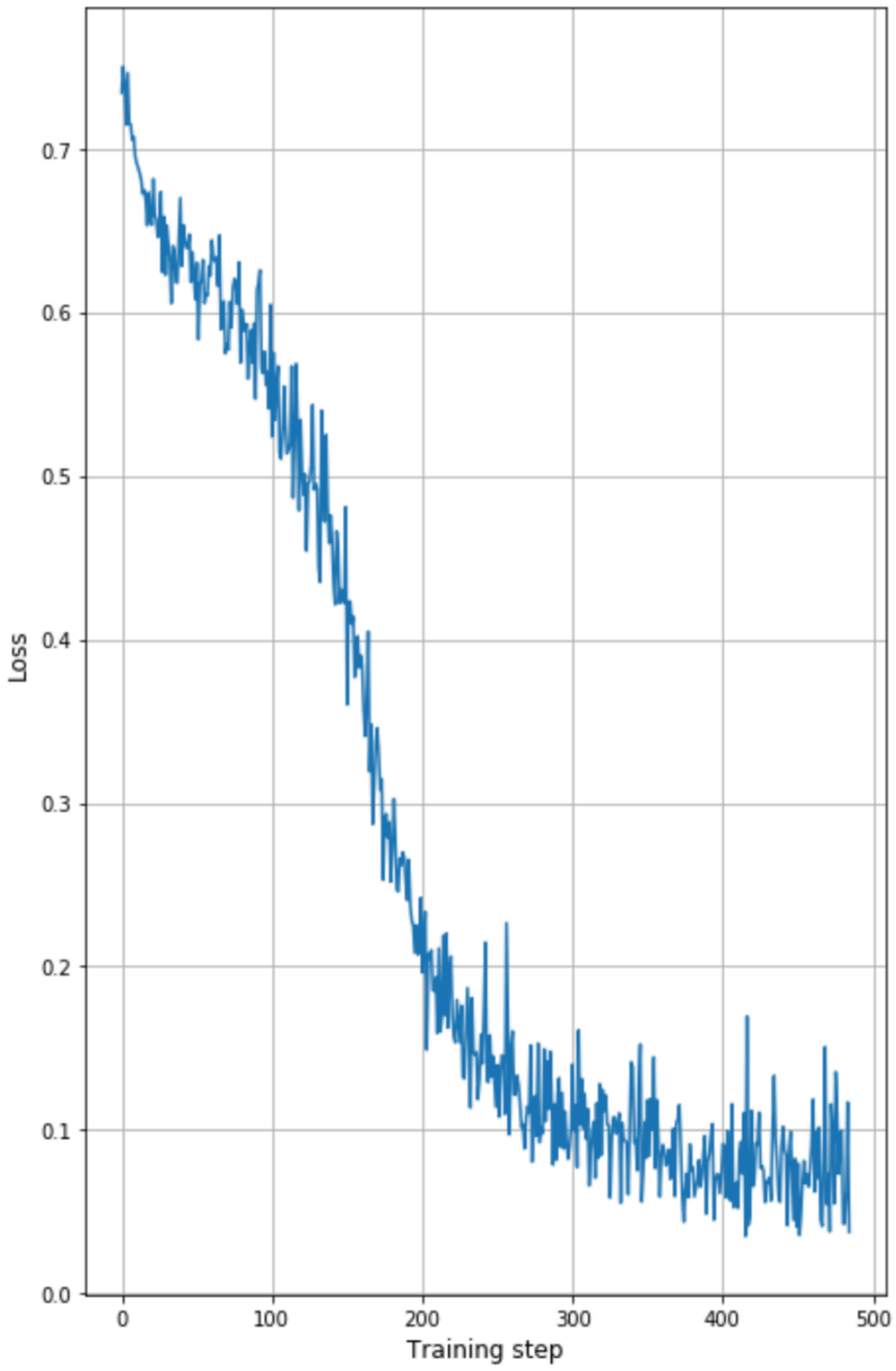
Tricks in Deep Learning

Boris Zubarev

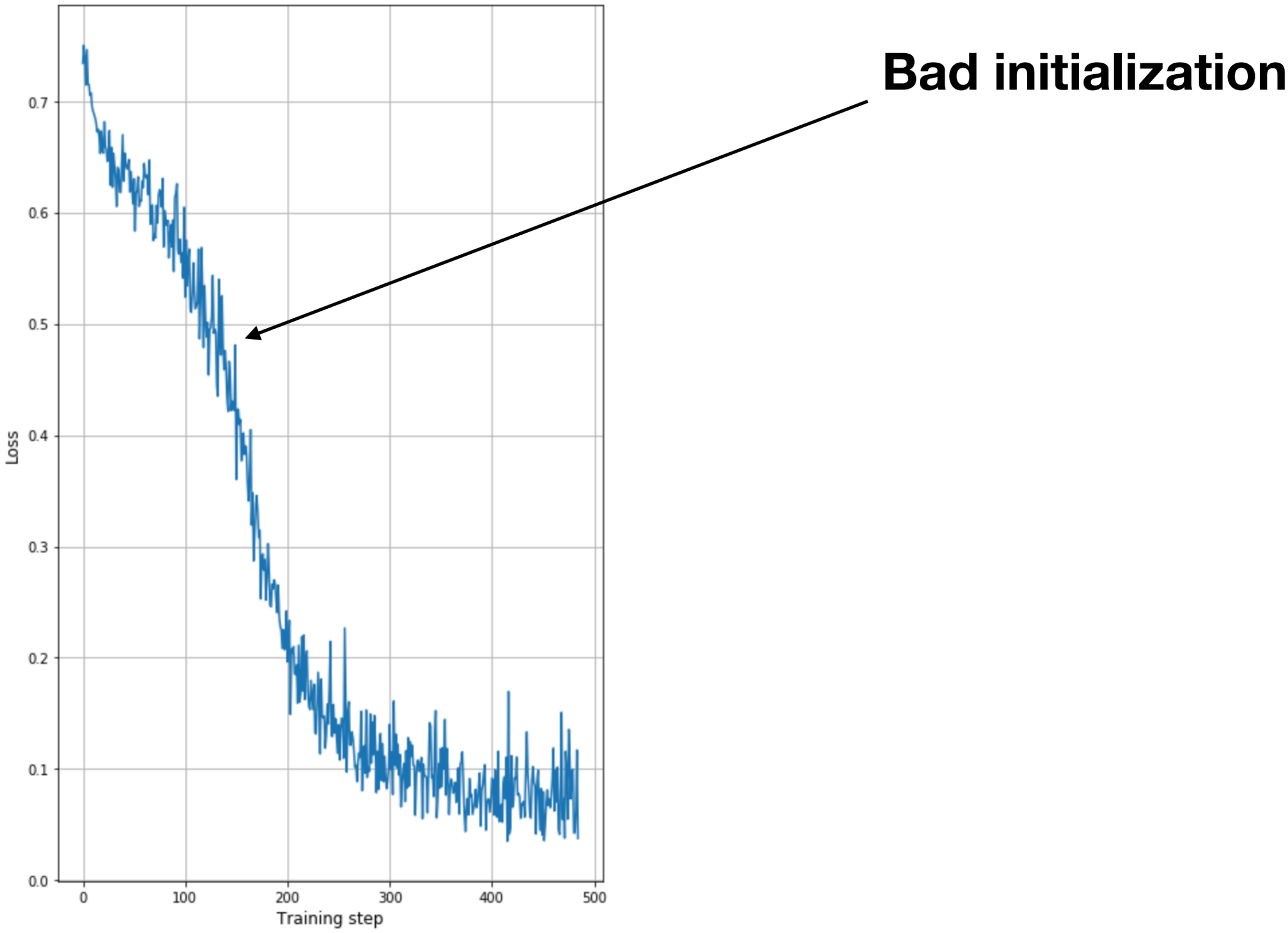
@bobazooba

Initialization

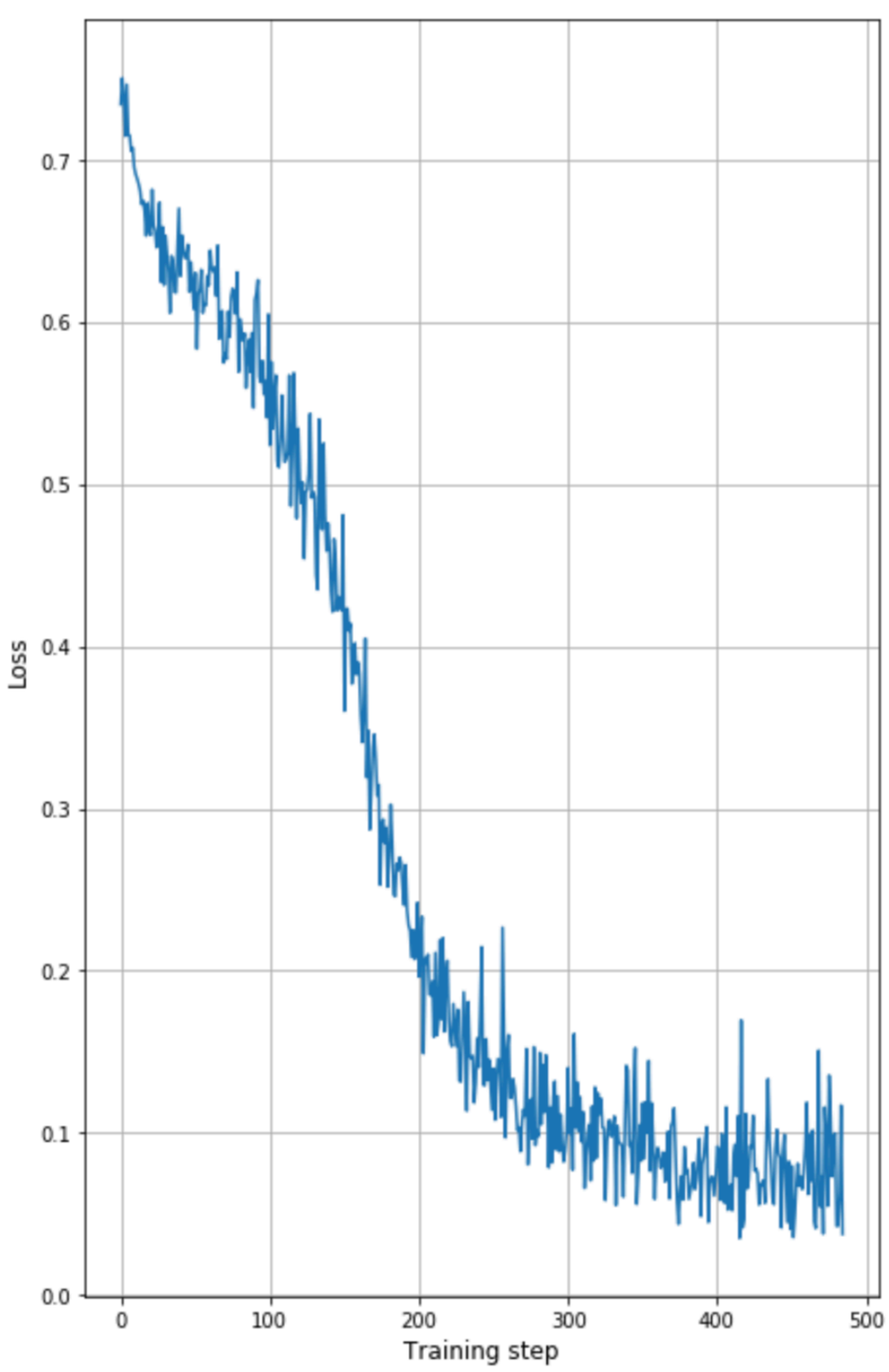
Initialization

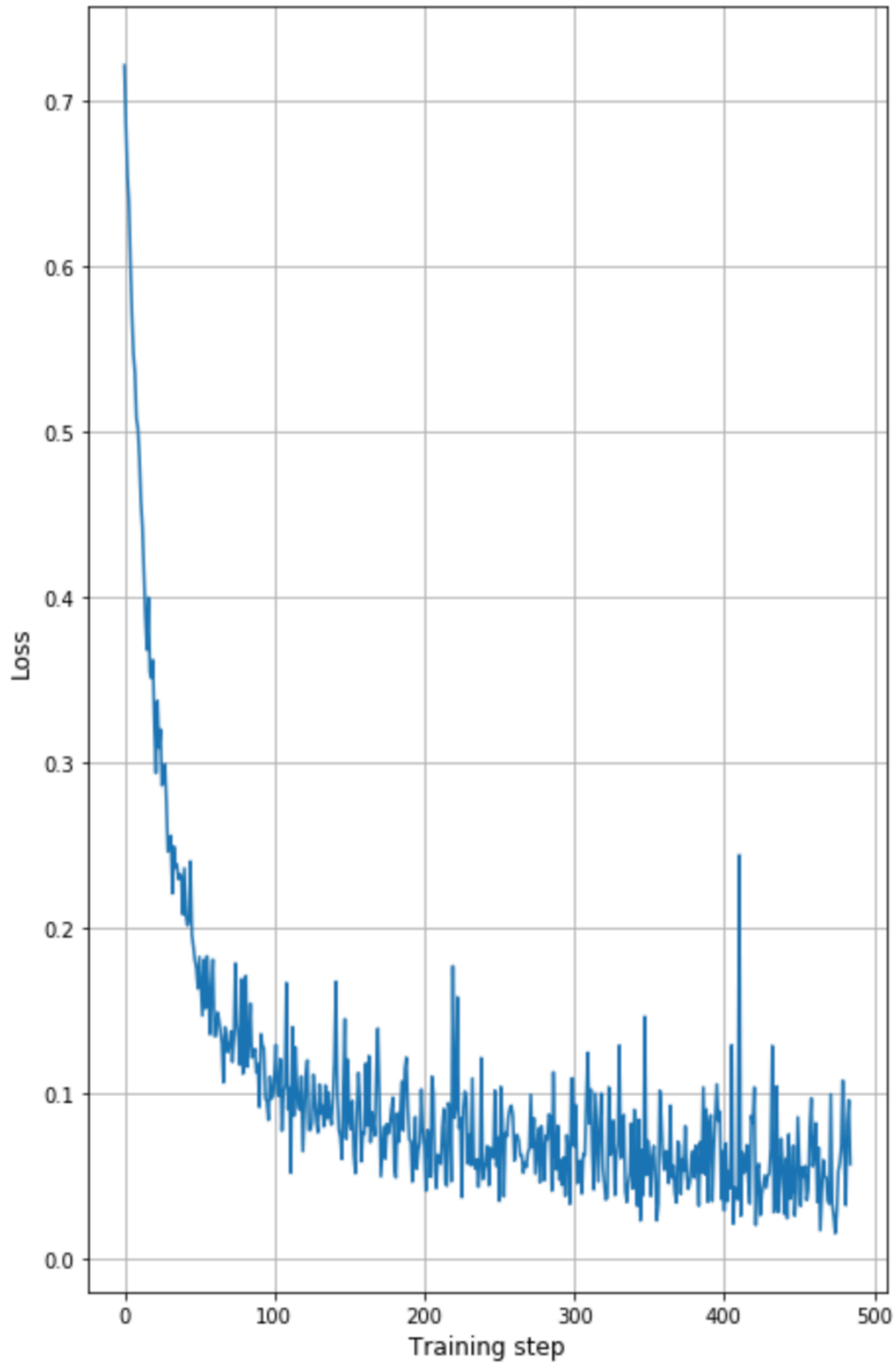


Initialization



Initialization



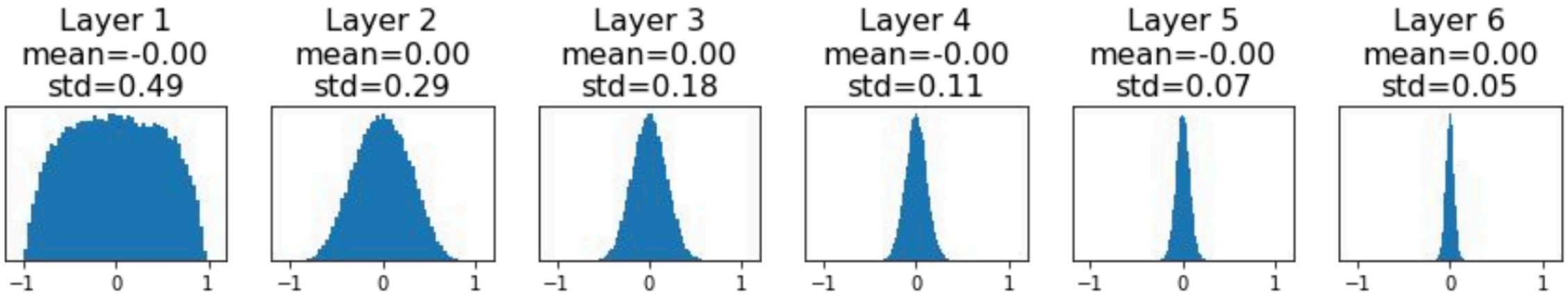


Initialization

weight = np.random.rand(in_features, out_features) * 0.01

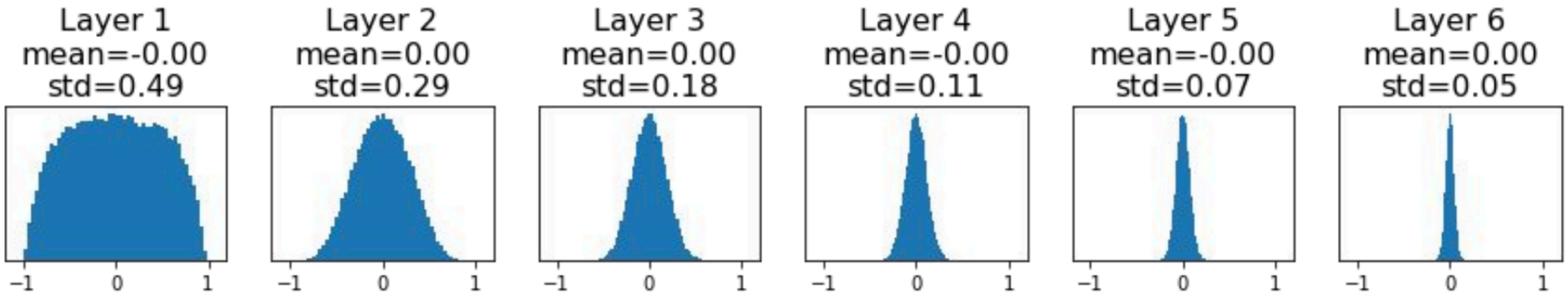
Initialization

weight = np.random.rand(in_features, out_features) * 0.01

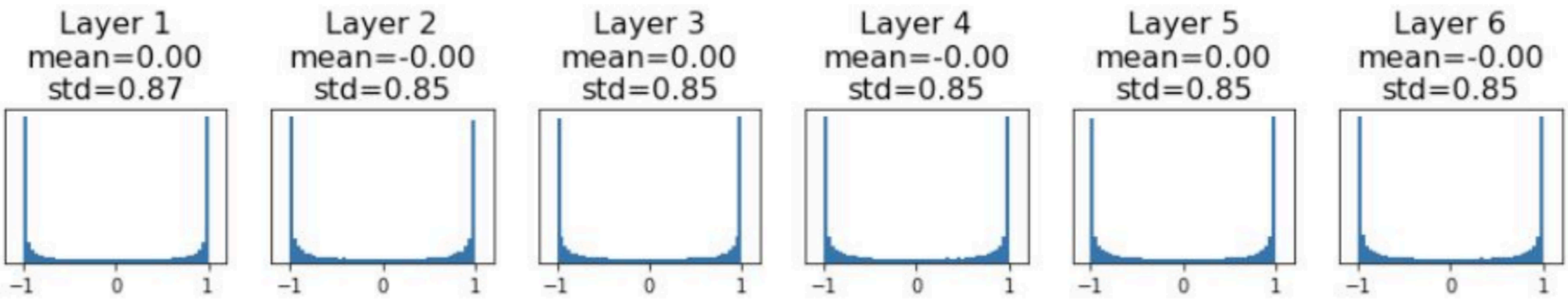


Initialization

weight = np.random.rand(in_features, out_features) * 0.01

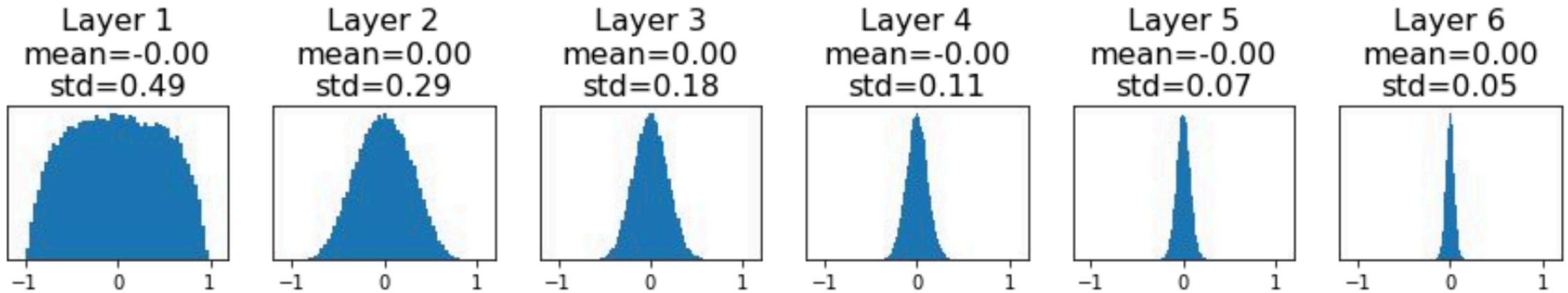


weight = np.random.rand(in_features, out_features) * 0.05

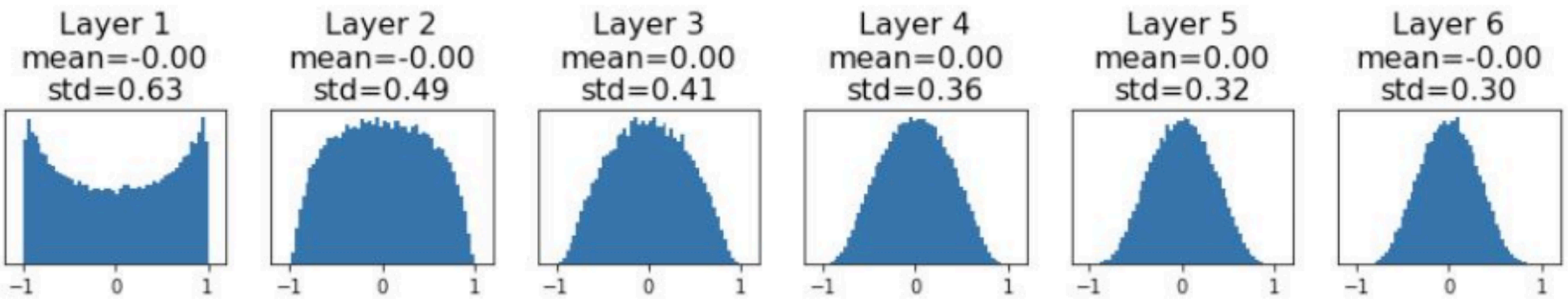


Initialization

weight = np.random.rand(in_features, out_features) * 0.01

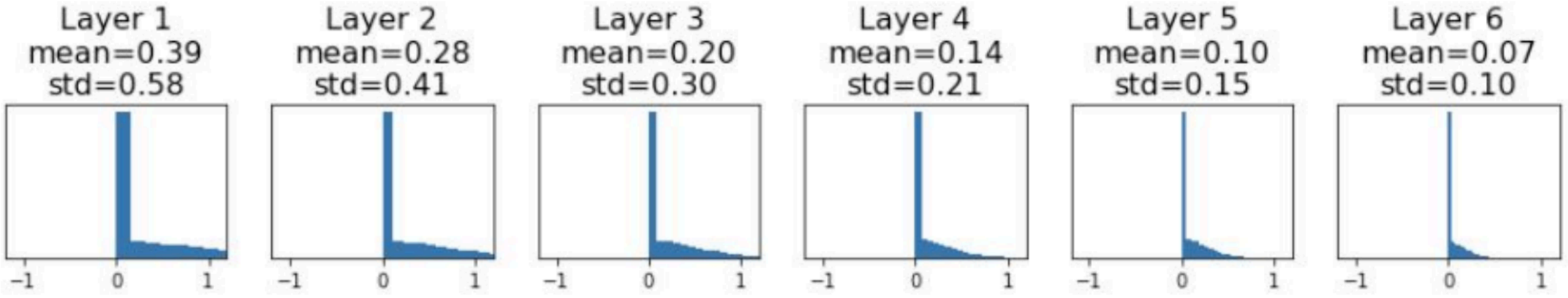


weight = np.random.rand(in_features, out_features) / sqrt(in_features)



Initialization

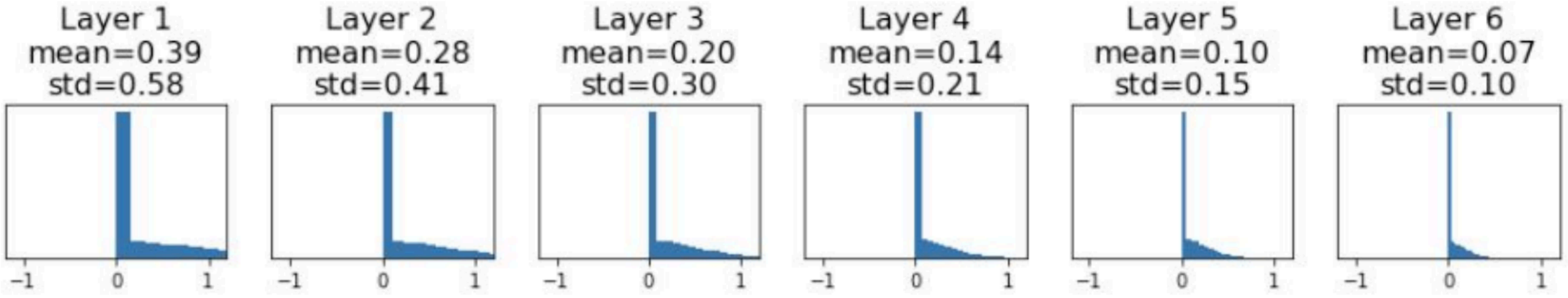
weight = np.random.rand(in_features, out_features) / sqrt(in_features)



With ReLU

Initialization

weight = np.random.rand(in_features, out_features) / sqrt(in_features)



With ReLU

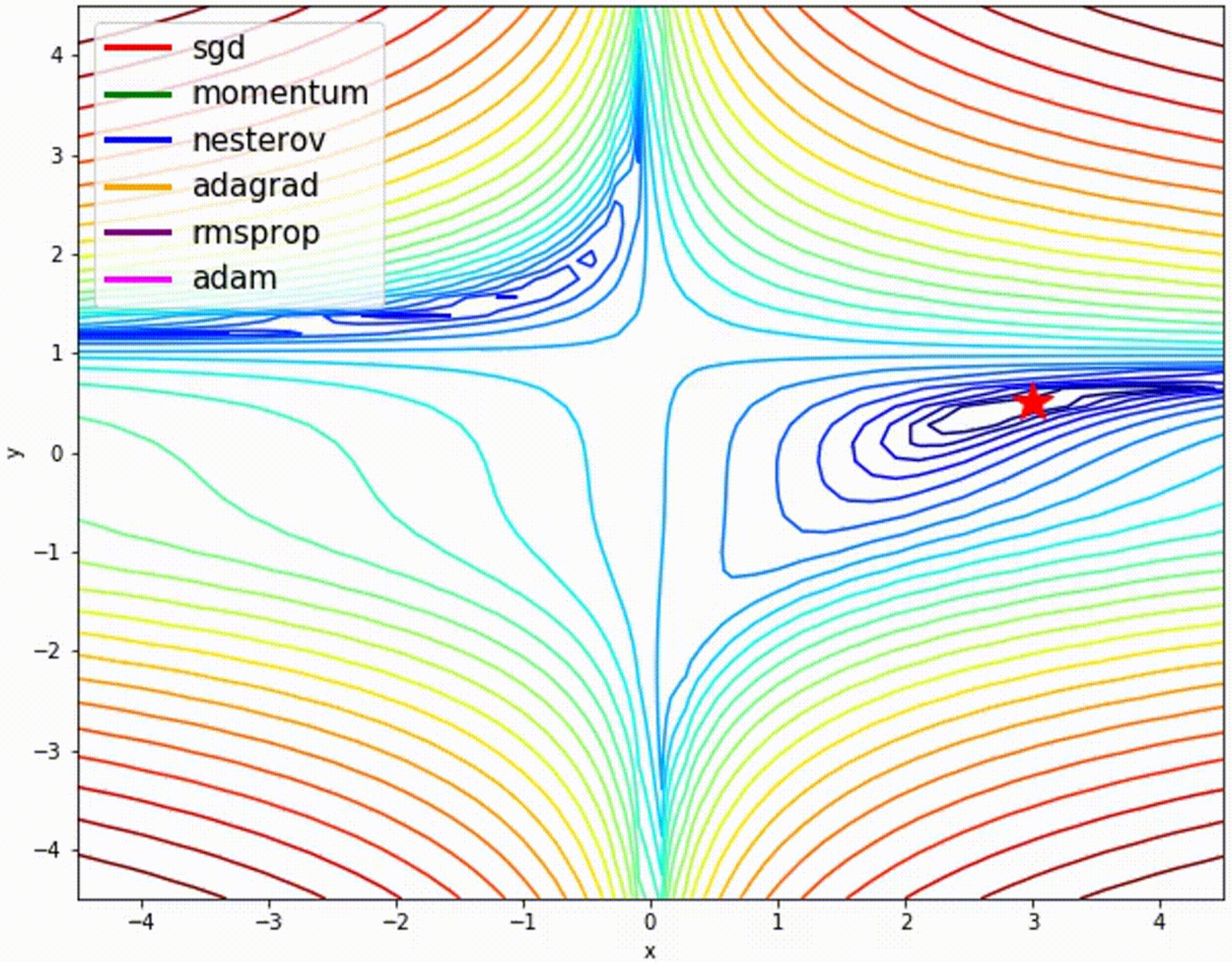
weight = np.random.rand(in_features, out_features) / sqrt(in_features / 2)

Initialization

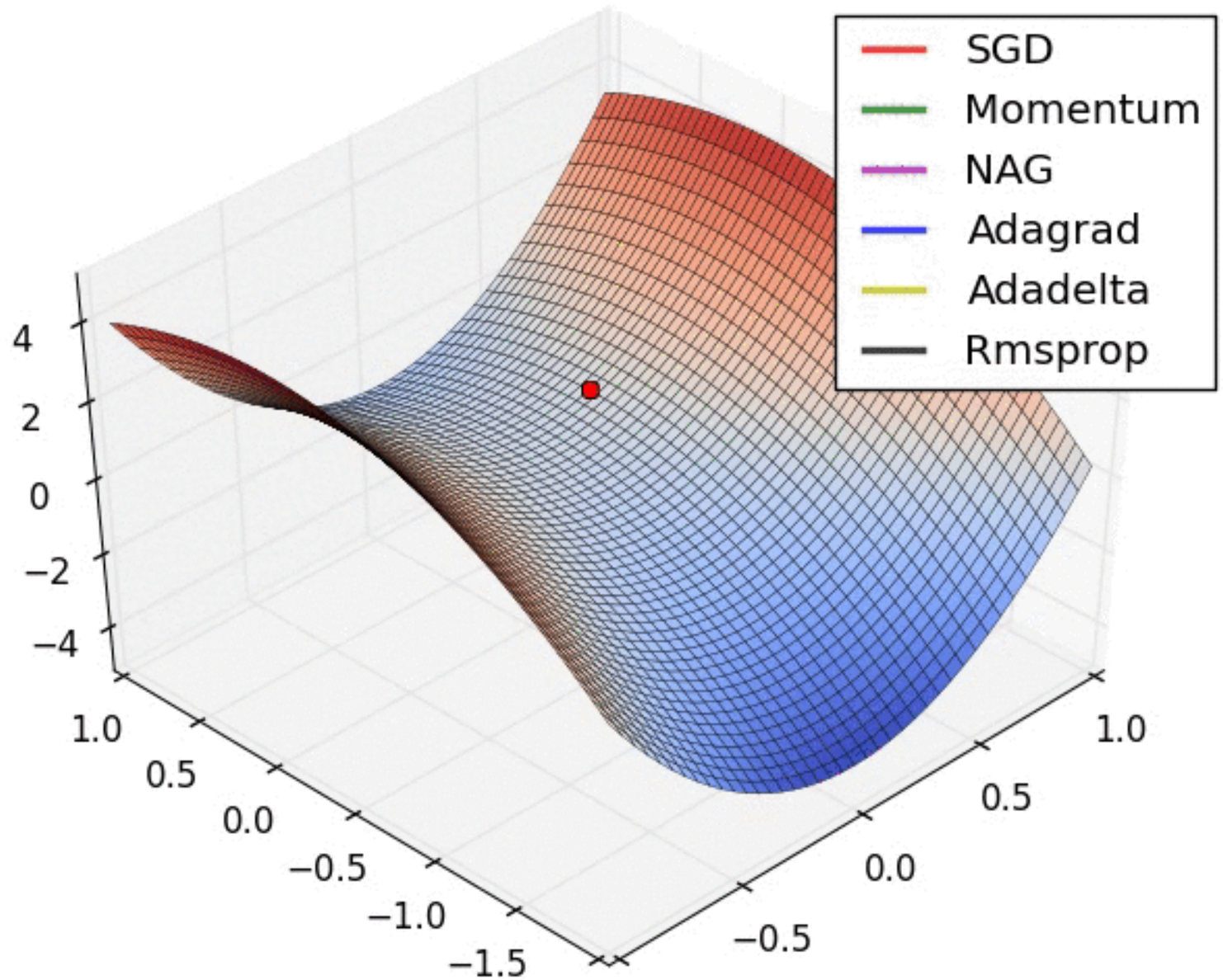
Variables

- **~Linear.weight** – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **~Linear.bias** – the learnable bias of the module of shape (out_features) . If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

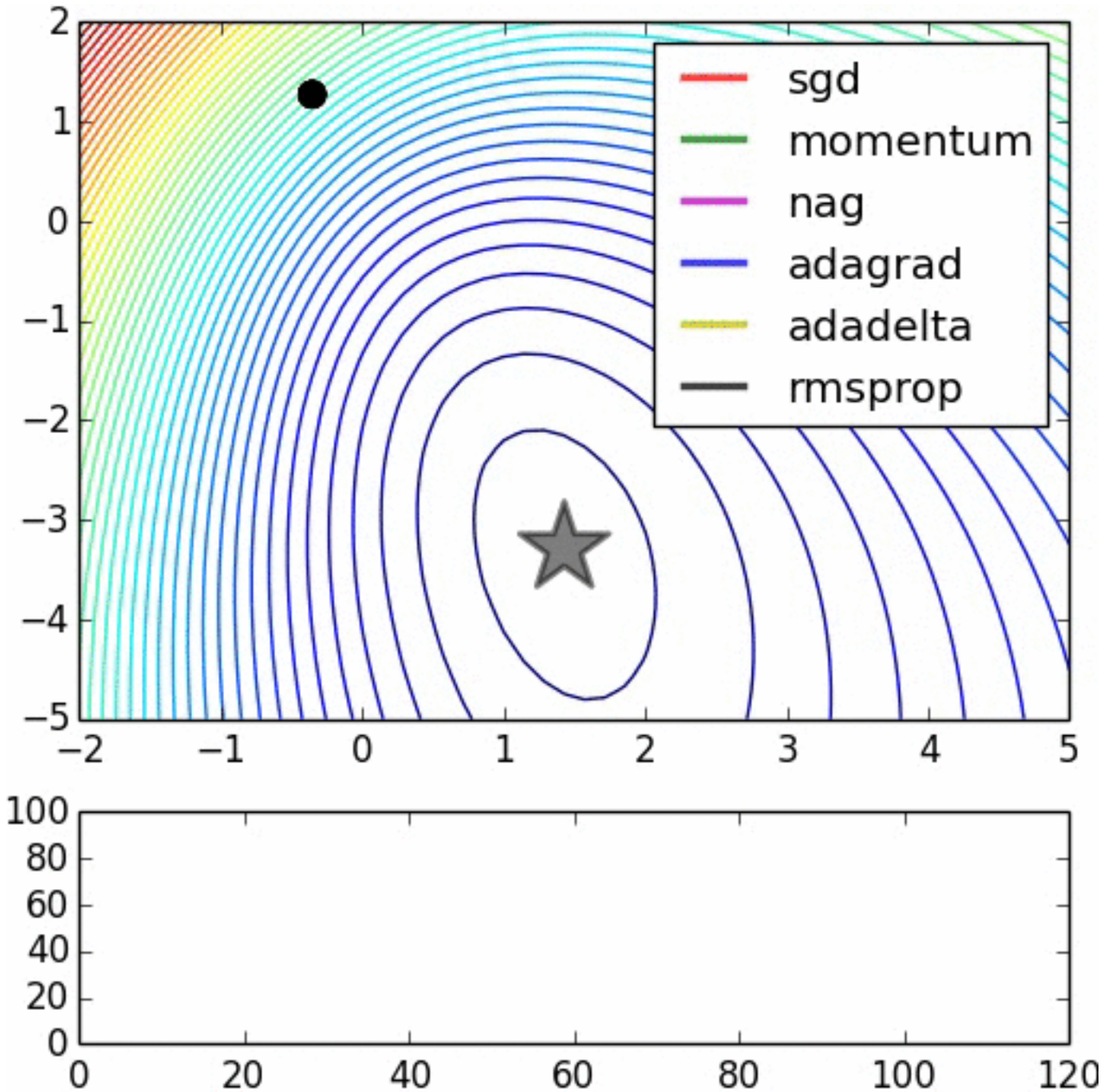
Optimizers


sgd
momentum
nesterov
adagrad
rmsprop
adam
y
x
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3
4

Optimizers


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5

Optimizers


sgd
momentum
nag
adagrad
adadelta
rmsprop
2
1
0
−1
−2
−3
−4
−5
−2
−1
0
1
2
3
4
5
100
80
60
40
20
0
0
20
40
60
80
100
120

Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

Momentum

Learning rate

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

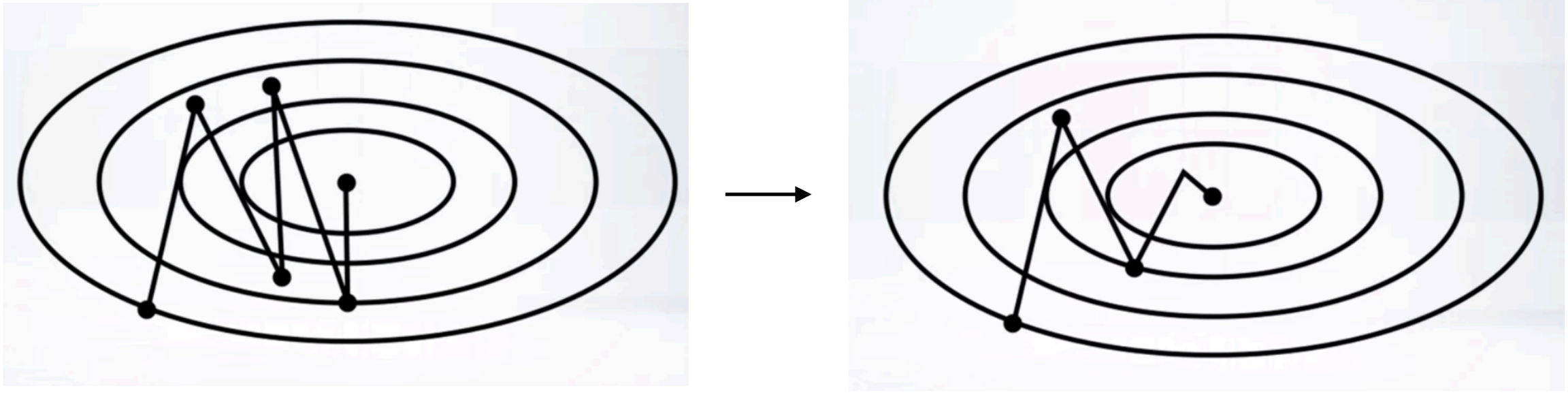
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

Momentum

SGD

SGD + Momentum



Nesterov Momentum

Momentum

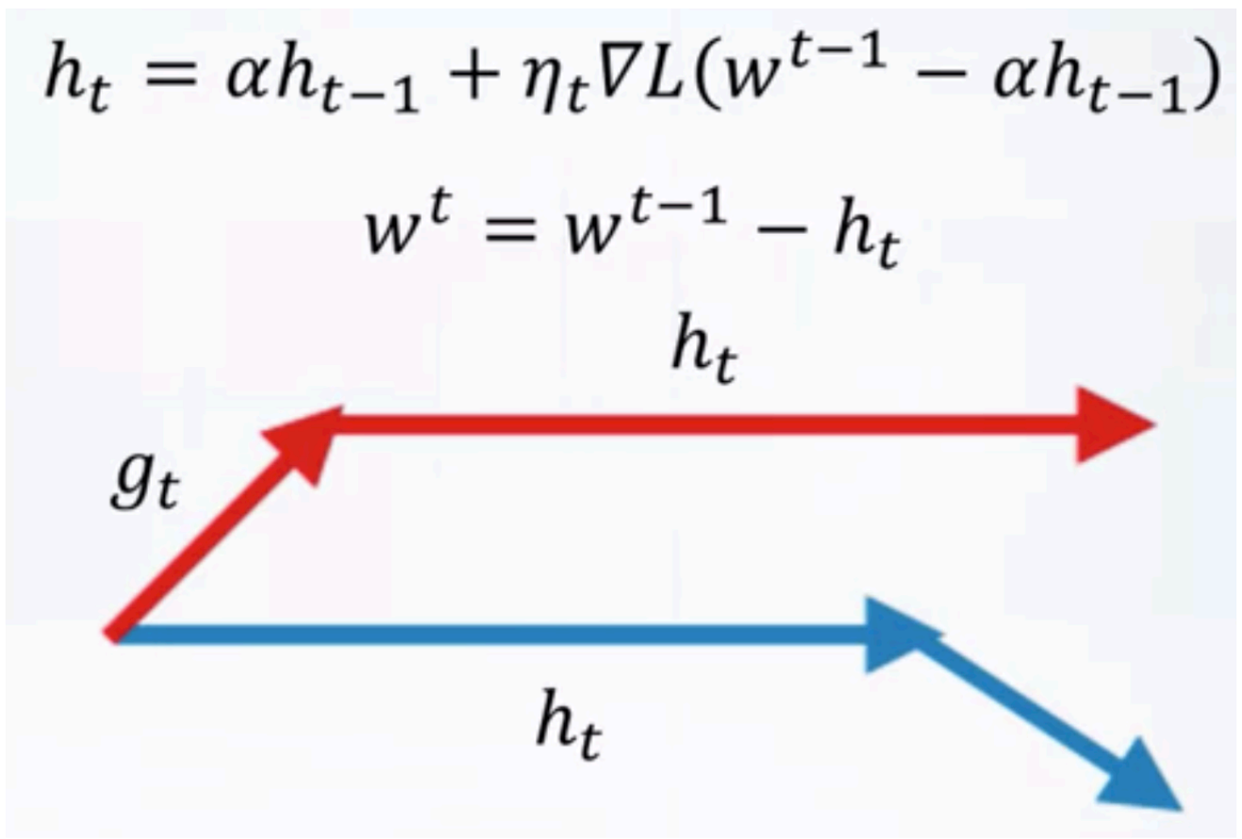
$$h_t = \alpha h_{t-1} + \eta_t g_t$$

$$w^t = w^{t-1} - h_t$$

Nesterov Momentum

$$h_t = \alpha h_{t-1} + \eta_t \nabla L(w^{t-1} - \alpha h_{t-1})$$

$$w^t = w^{t-1} - h_t$$



Nesterov Momentum

Momentum

$$h_t = \alpha h_{t-1} + \eta_t g_t$$

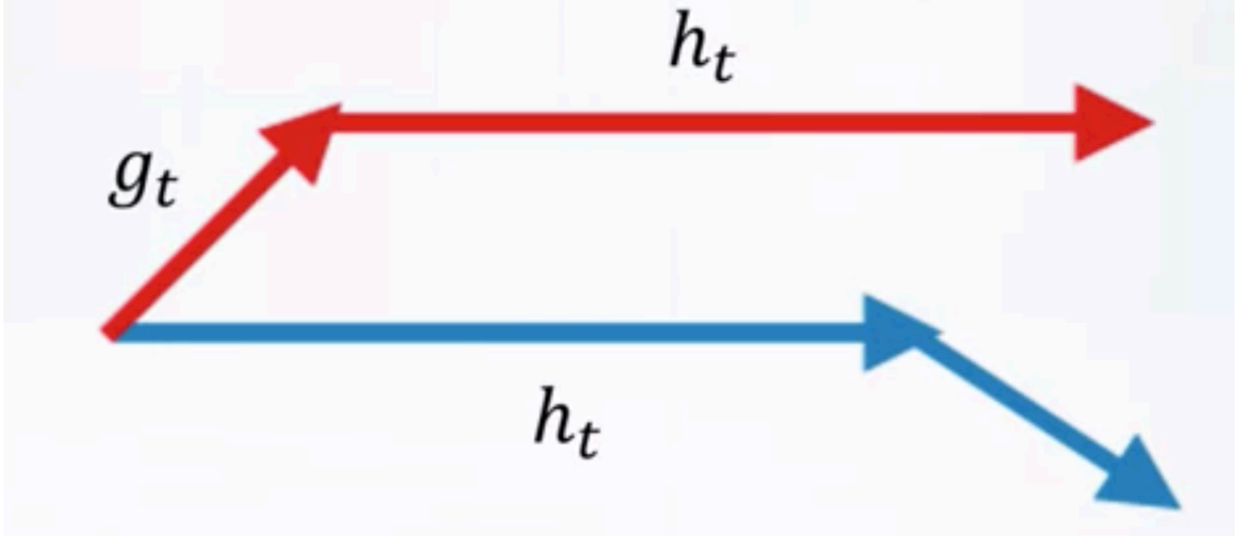
$$w^t = w^{t-1} - h_t$$

Learning rate

Nesterov Momentum

$$h_t = \alpha h_{t-1} + \eta_t \nabla L(w^{t-1} - \alpha h_{t-1})$$

$$w^t = w^{t-1} - h_t$$



Nesterov Momentum

Momentum

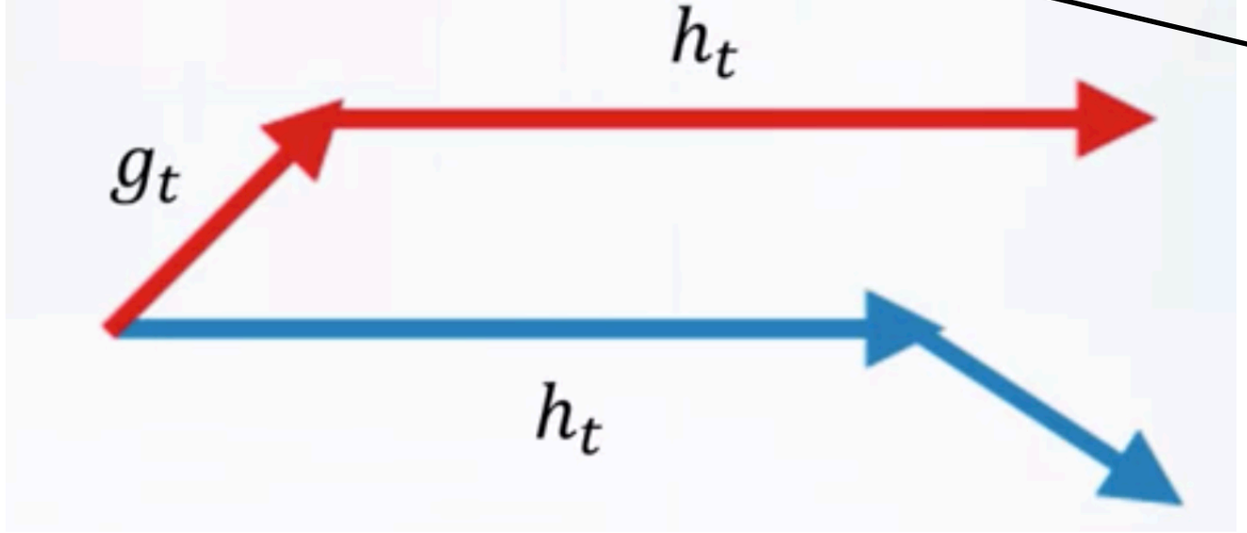
$$h_t = \alpha h_{t-1} + \eta_t g_t$$

$$w^t = w^{t-1} - h_t$$

Nesterov Momentum

$$h_t = \alpha h_{t-1} + \eta_t \nabla L(w^{t-1} - \alpha h_{t-1})$$

$$w^t = w^{t-1} - h_t$$



alpha ~= 0.9

AdaGrad

$$G_j^t = G_j^{t-1} + g_{tj}^2$$

$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{G_j^t + \epsilon}} g_{tj}$$

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

AdaGrad

$$G_j^t = G_j^{t-1} + g_{tj}^2$$

$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{G_j^t + \epsilon}} g_{tj}$$

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Learning rate goes to zero

AdaGrad

$$G_j^t = G_j^{t-1} + g_{tj}^2$$

$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{G_j^t + \epsilon}} g_{tj}$$

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Learning rate goes to zero

Progress along «steep» directions is damped

Progress along «flat» directions is accelerated

RMSprop

$$G_j^t = \alpha G_j^{t-1} + (1-\alpha) g_{tj}^2$$

$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{G_j^t + \epsilon}} g_{tj}$$

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSprop

$$G_j^t = \alpha G_j^{t-1} + (1-\alpha) g_{tj}^2$$

$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{G_j^t + \epsilon}} g_{tj}$$

alpha ~= 0.9

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSprop

$$G_j^t = \alpha G_j^{t-1} + (1-\alpha) g_{tj}^2$$

$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{G_j^t + \epsilon}} g_{tj}$$

Learning rate

alpha ~= 0.9

AdaGrad

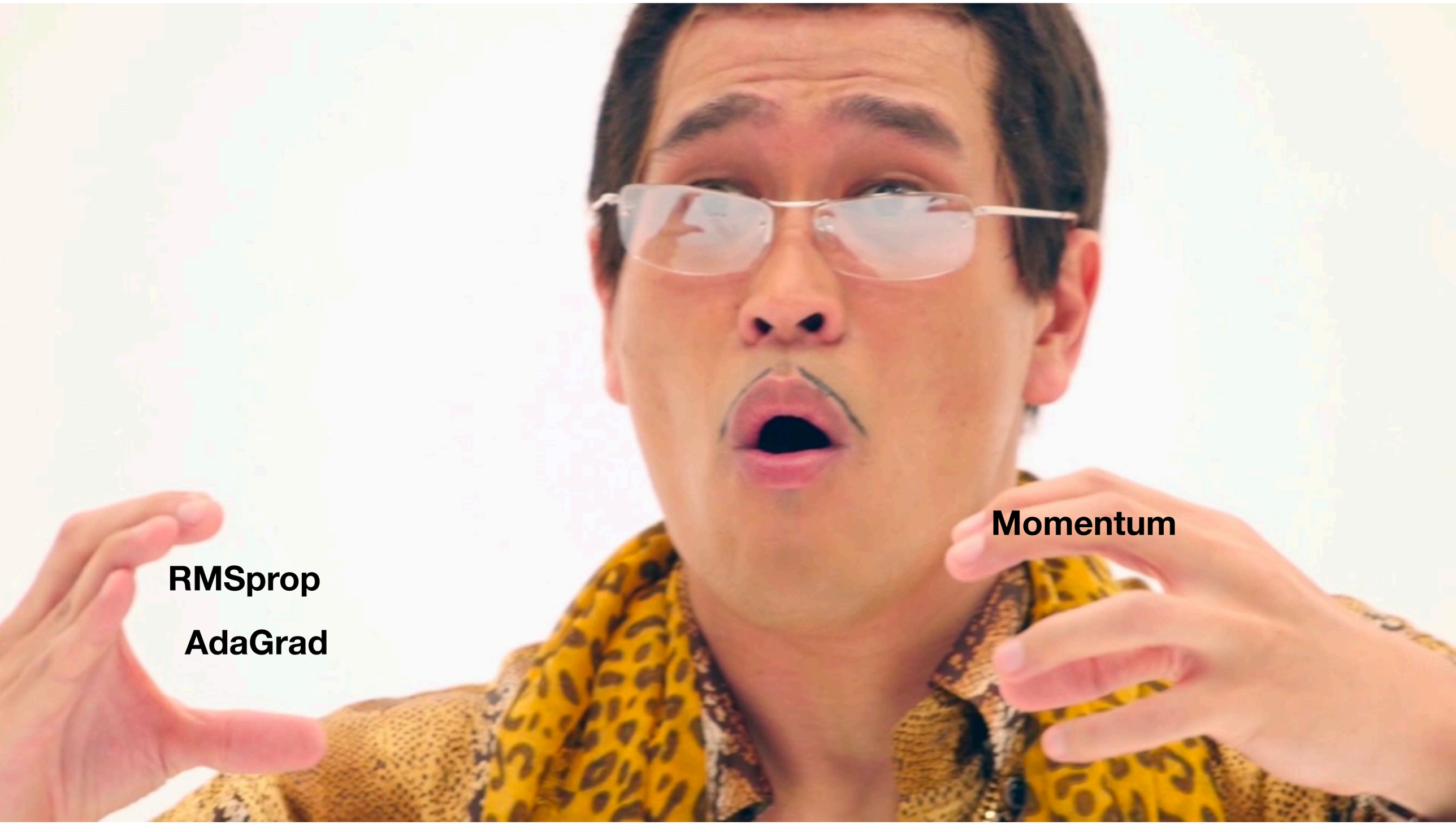
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Adam

Adam



Adam

$$v_j^t = \frac{\beta_2 v_j^{t-1} + (1-\beta_2) g_{tj}^2}{1-\beta_2^t}$$

$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{v_j^t + \epsilon}} g_{tj}$$

Adam

$$v_j^t = \frac{\beta_2 v_j^{t-1} + (1-\beta_2) g_{tj}^2}{1-\beta_2^t}$$

$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{v_j^t + \epsilon}}\, g_{tj}$$

↓

$$m_j^t = \frac{\beta_1 m_j^{t-1} + (1-\beta_1) g_{tj}}{1-\beta_1^t}$$

$$v_j^t = \frac{\beta_2 v_j^{t-1} + (1-\beta_2) g_{tj}^2}{1-\beta_2^t}$$

$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{v_j^t + \epsilon}}\, m_j^t$$

Adam

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

Adam

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Adam

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction
for the fact that
first and second moment
estimates start from zero

Adam

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

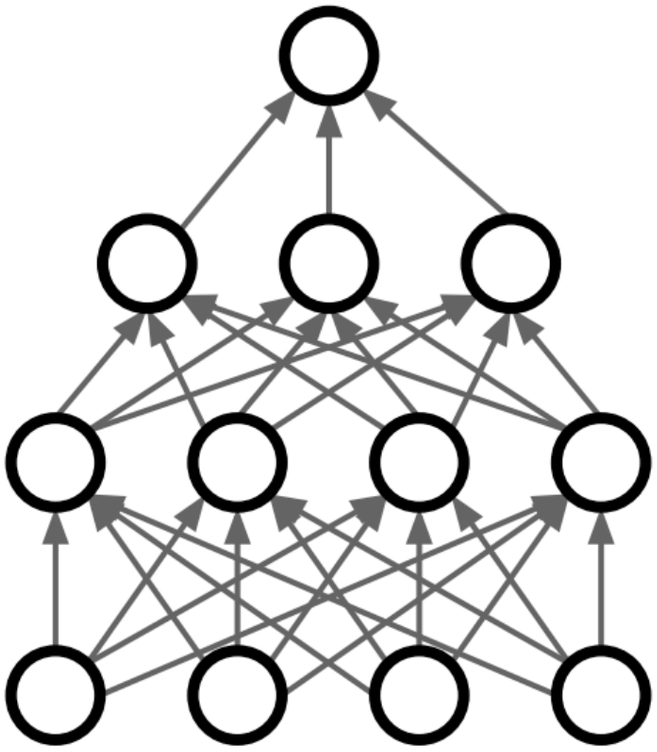
Bias correction

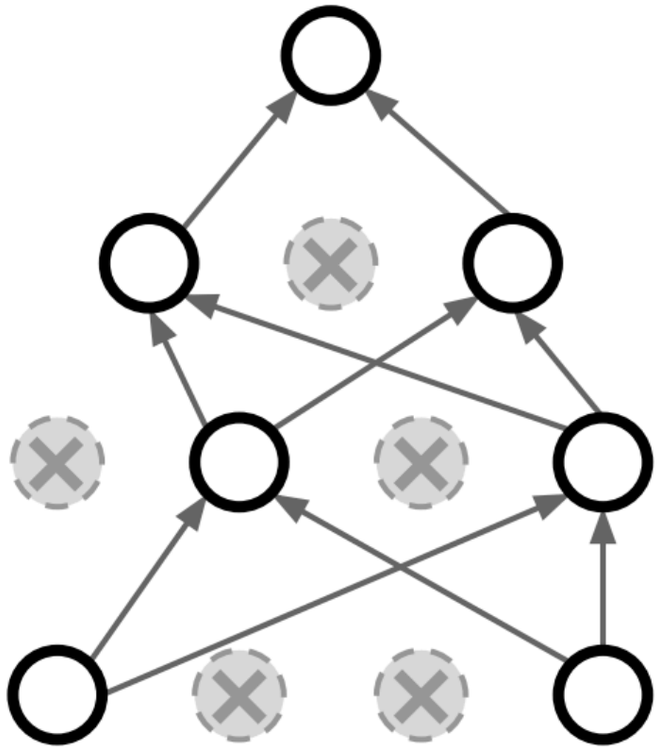
AdaGrad / RMSProp

Bias correction
for the fact that
first and second moment
estimates start from zero

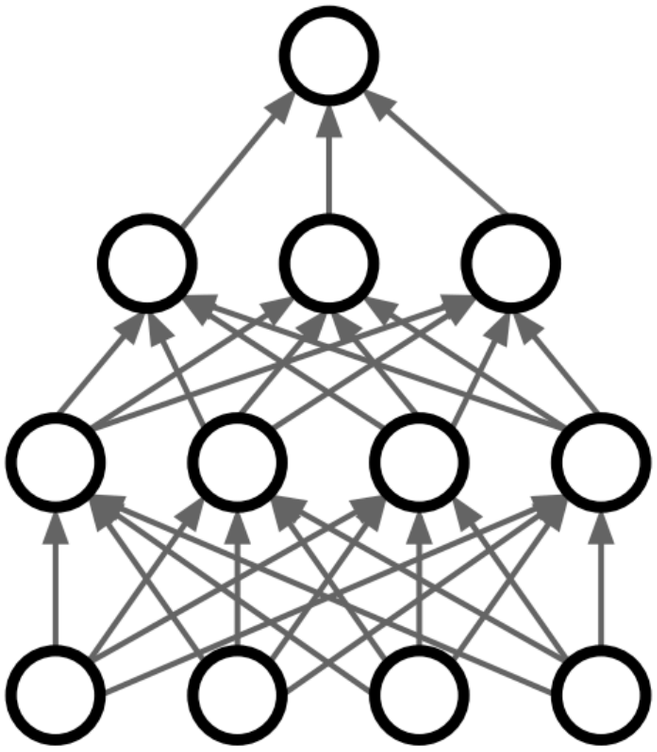
This optimizer is default choice

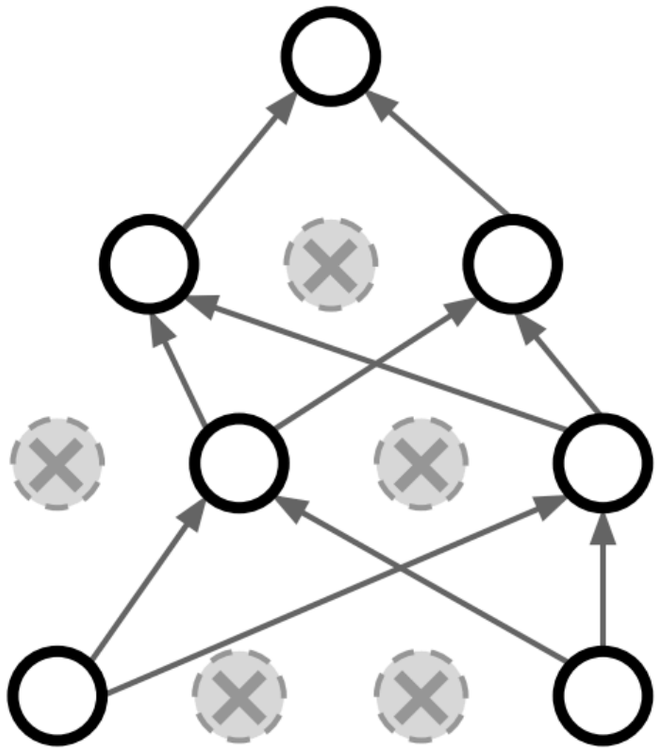
Dropout





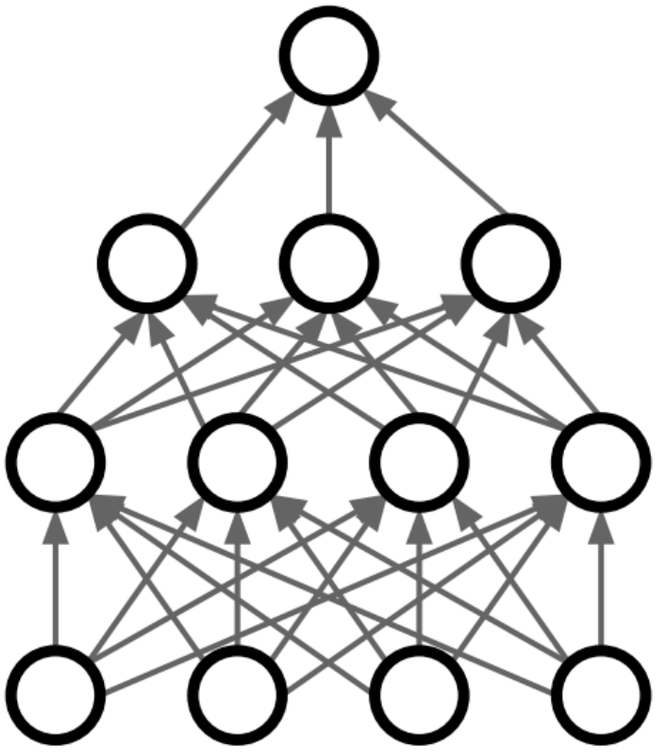
Dropout

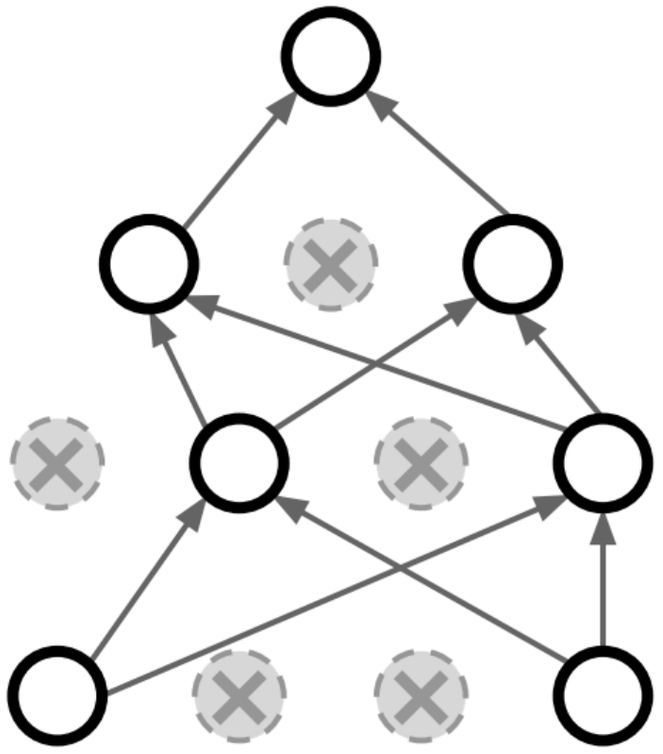





has an ear
has a tail
is furry
has claws
mischievous look
X
X
X
cat score

Dropout

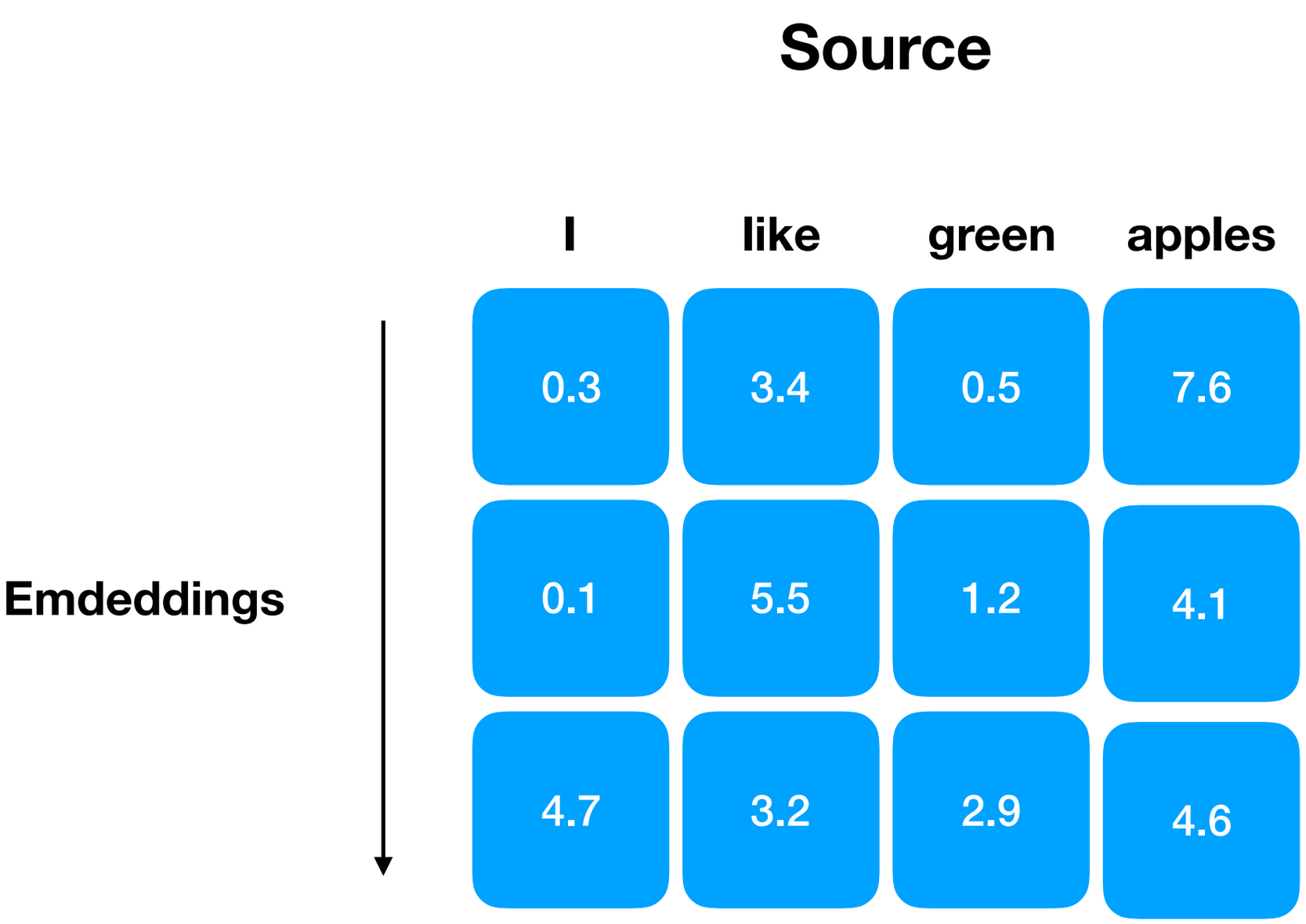




Ensemble of models with shared parameters

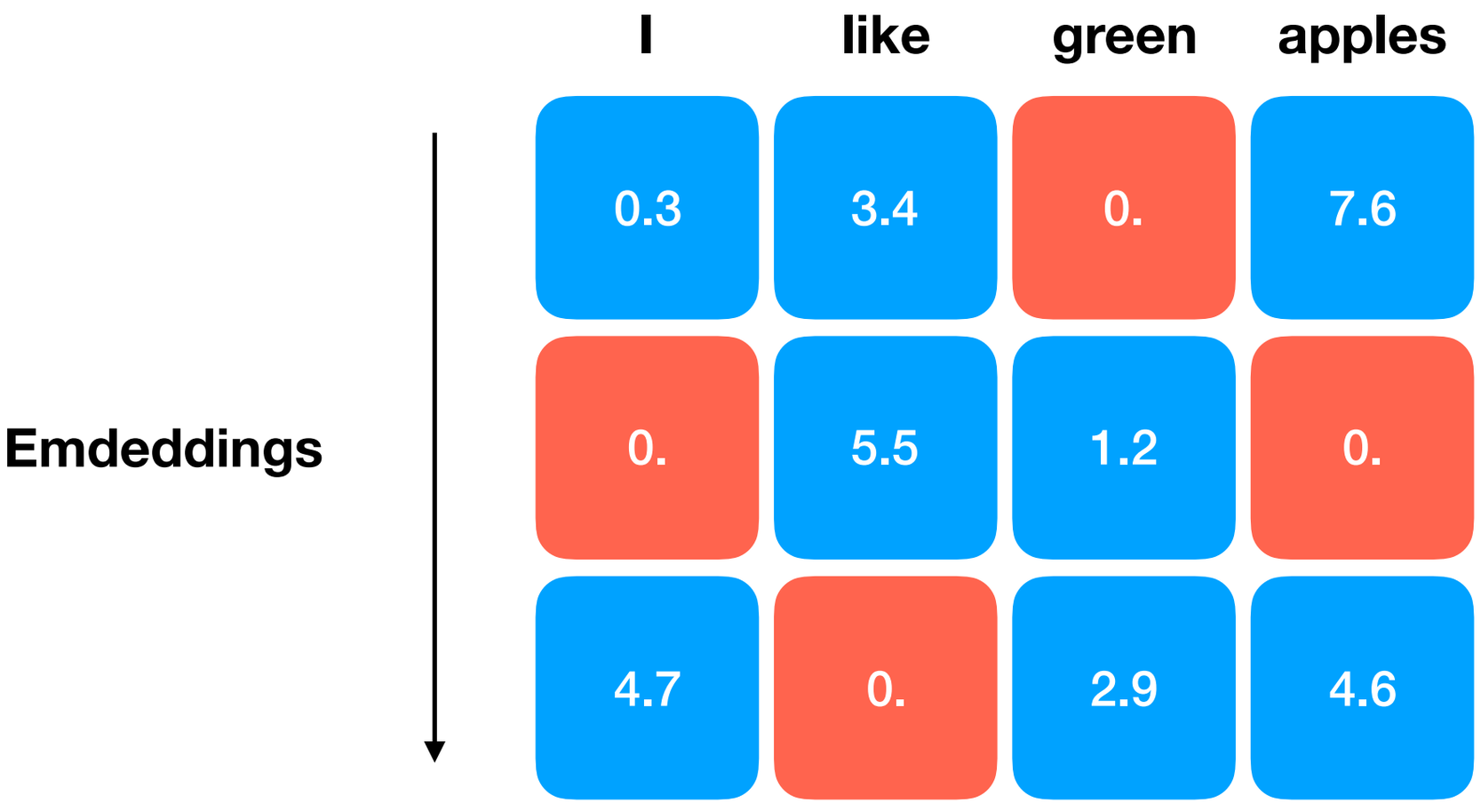
At test time, multiply by dropout probability

Dropout

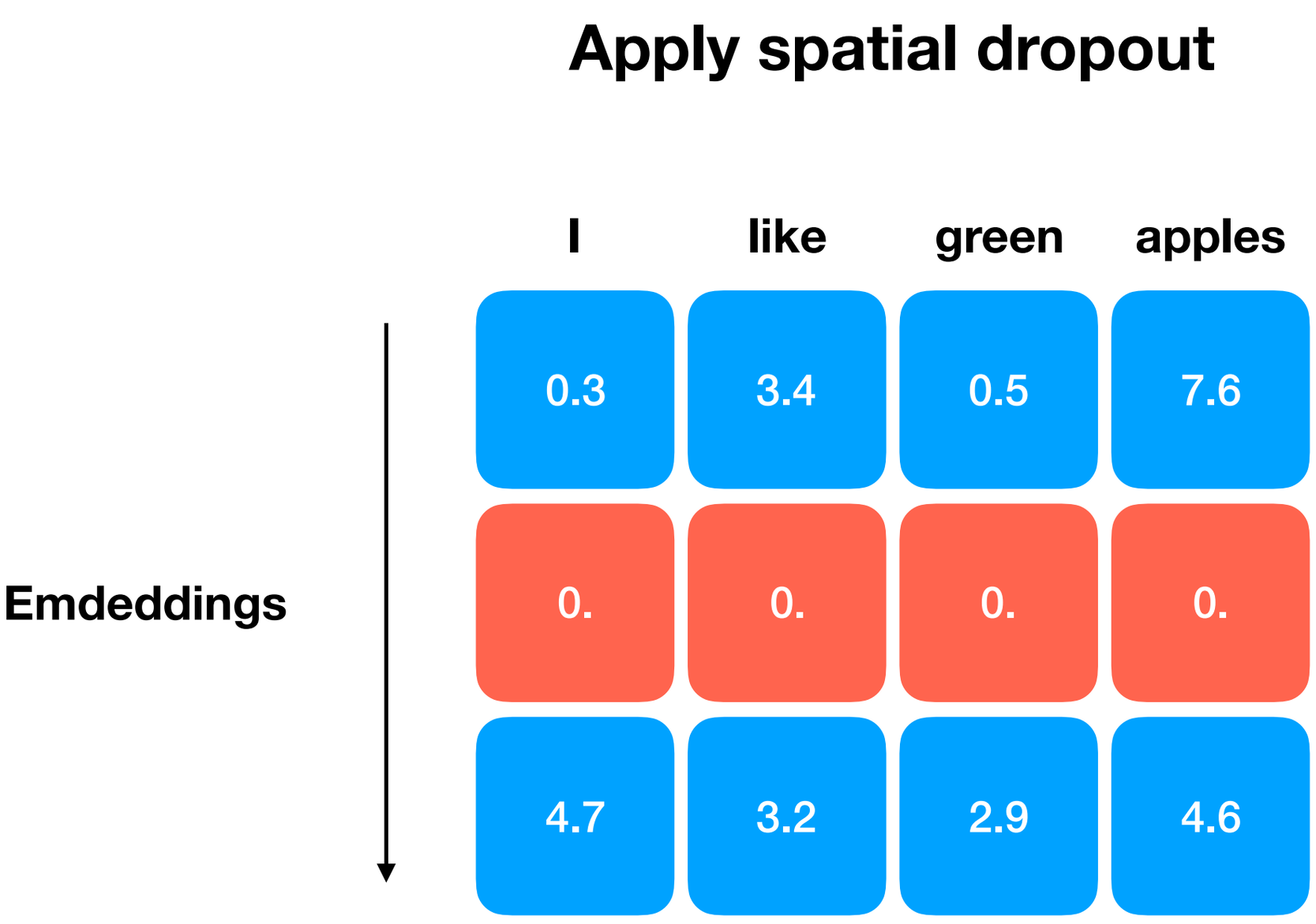

Source
I
like
green
apples
Emdeddings
0.3
3.4
0.5
7.6
0.1
5.5
1.2
4.1
4.7
3.2
2.9
4.6

Dropout

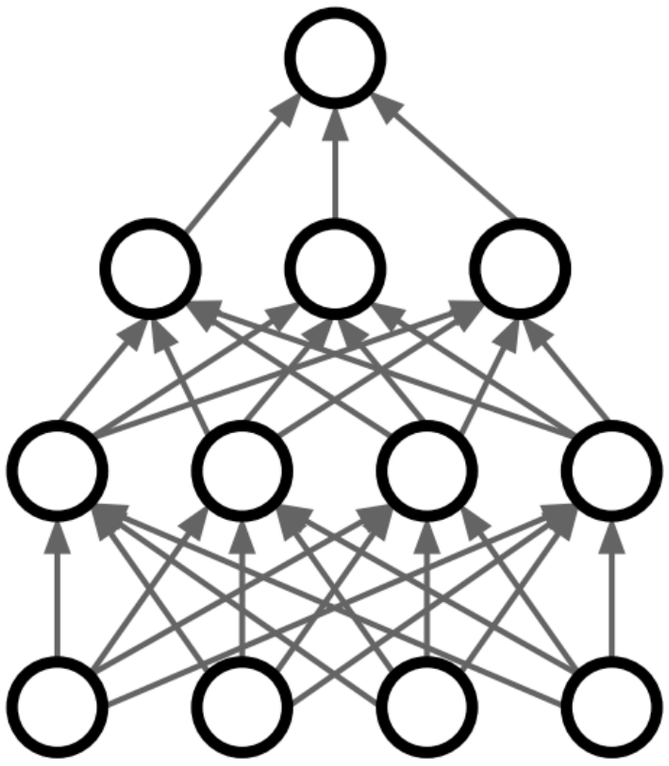
Apply dropout

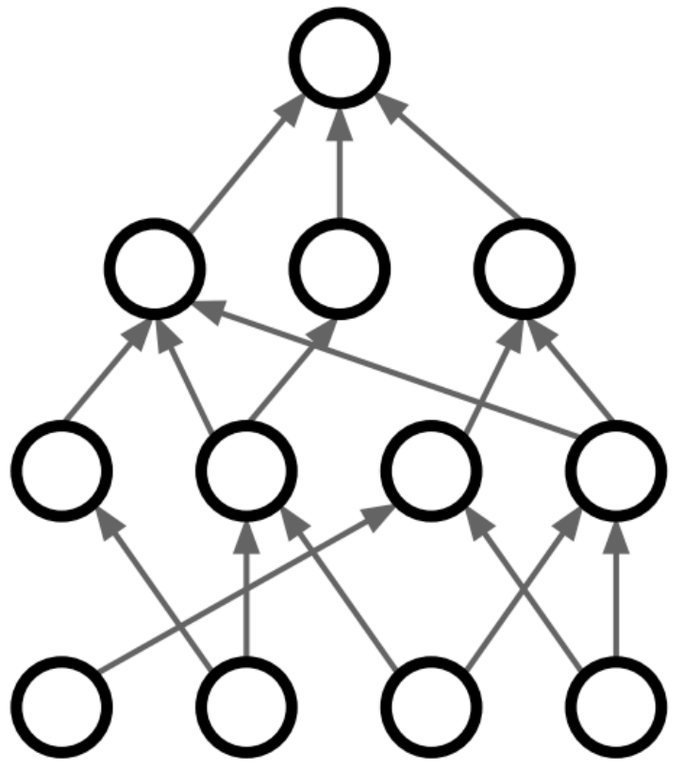


Spatial Dropout

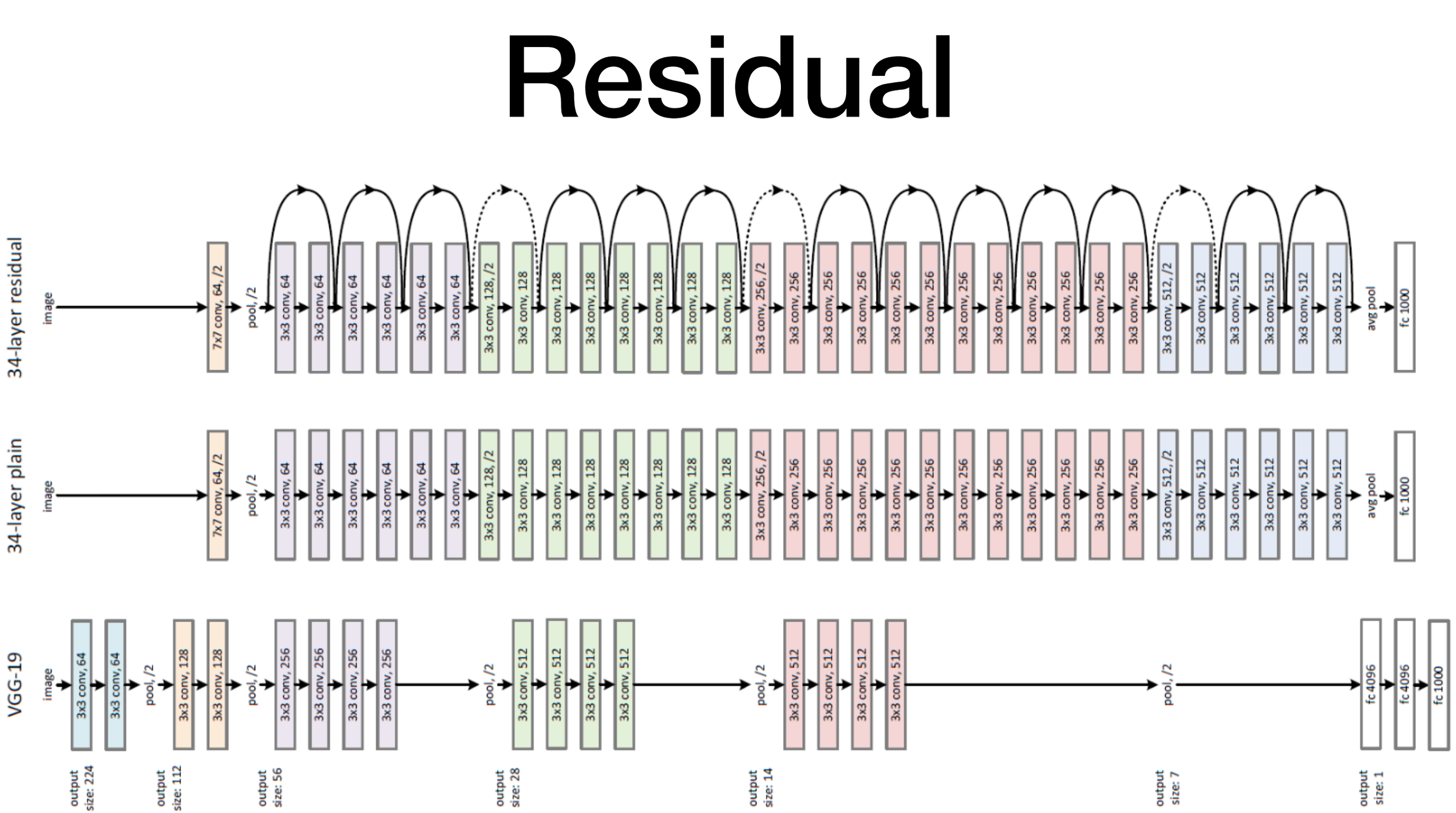


DropConnect



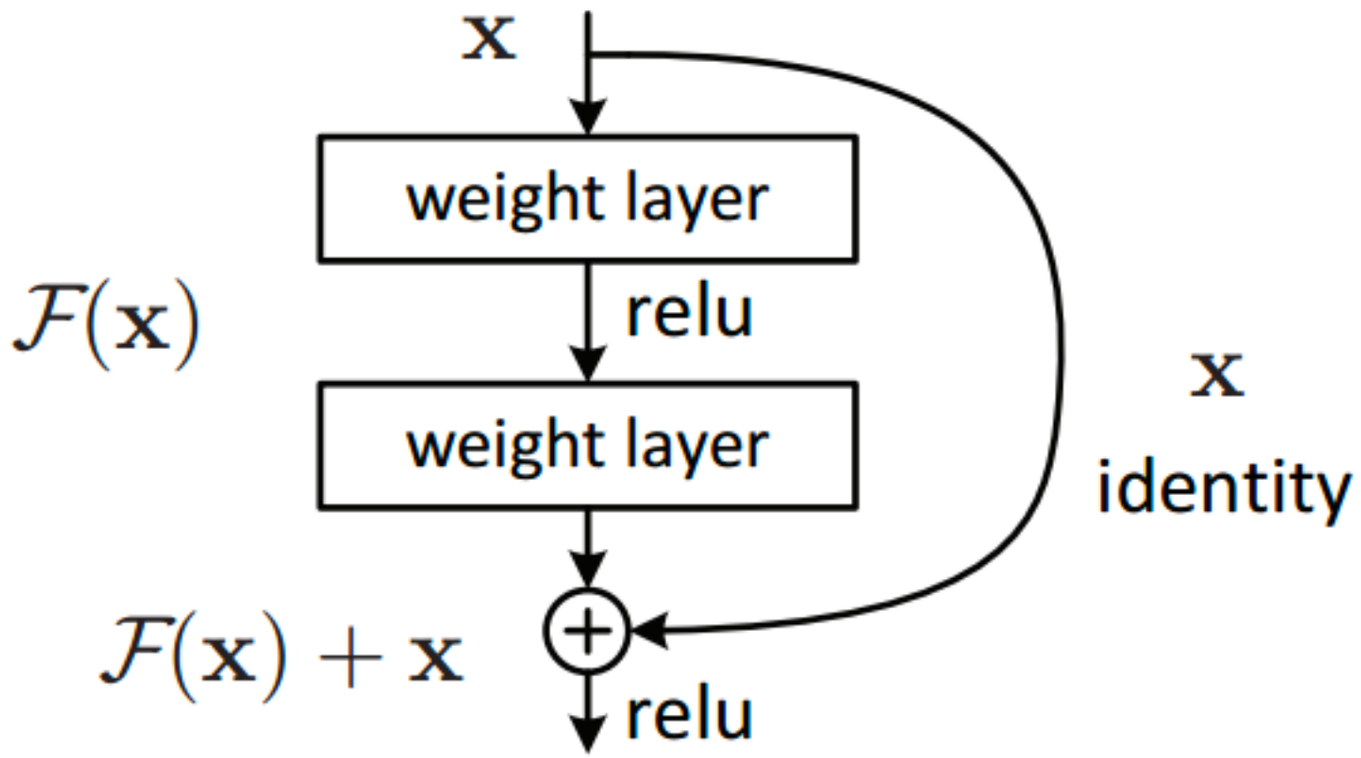


Residual

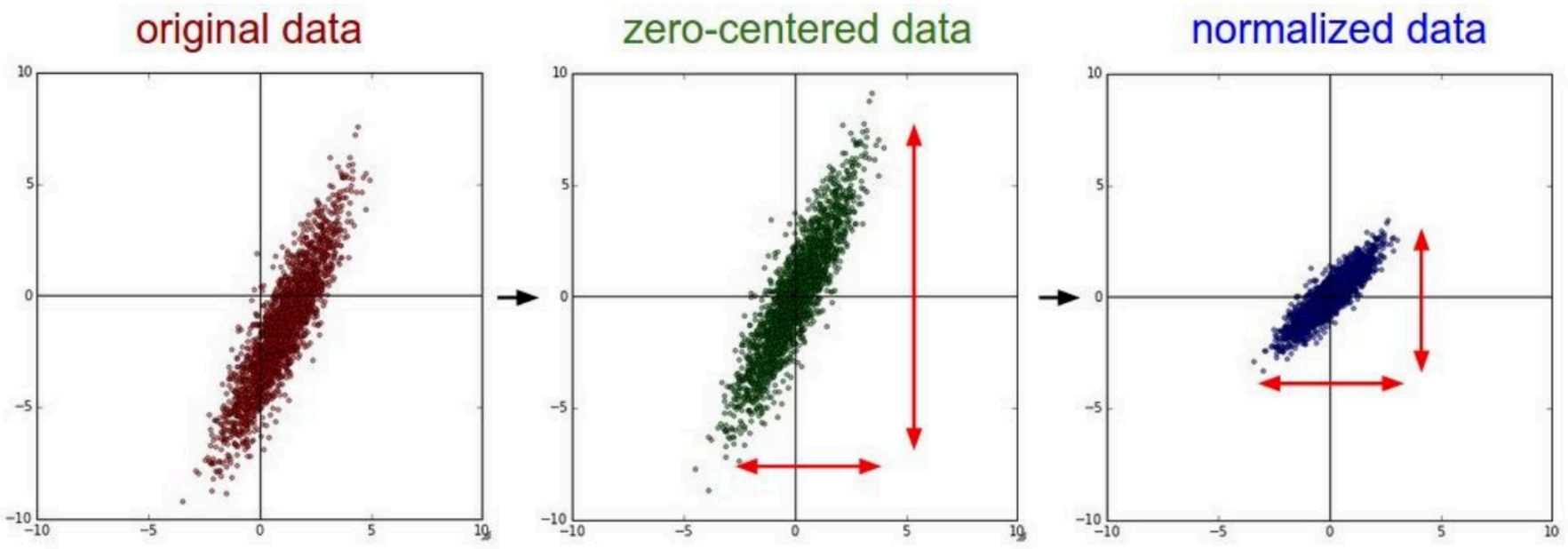


	plain	ResNet
18 layers	27.94	27.88
34 layers	28.54	**25.03**

Residual


x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

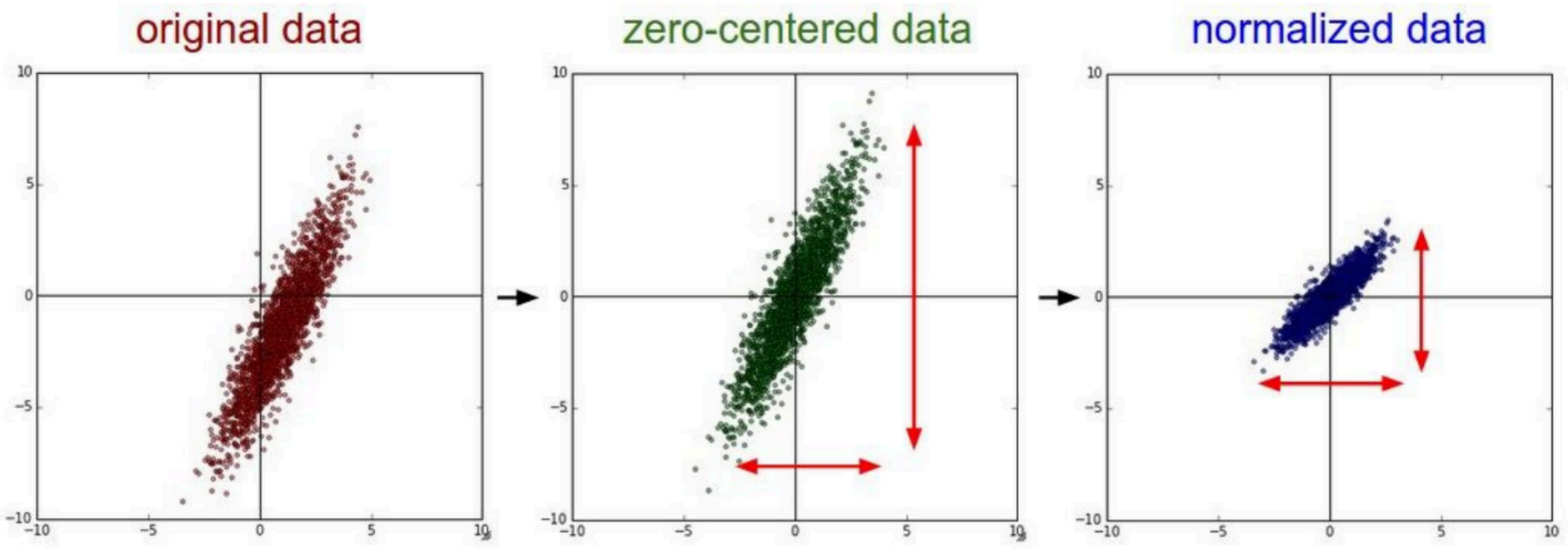
Normalization



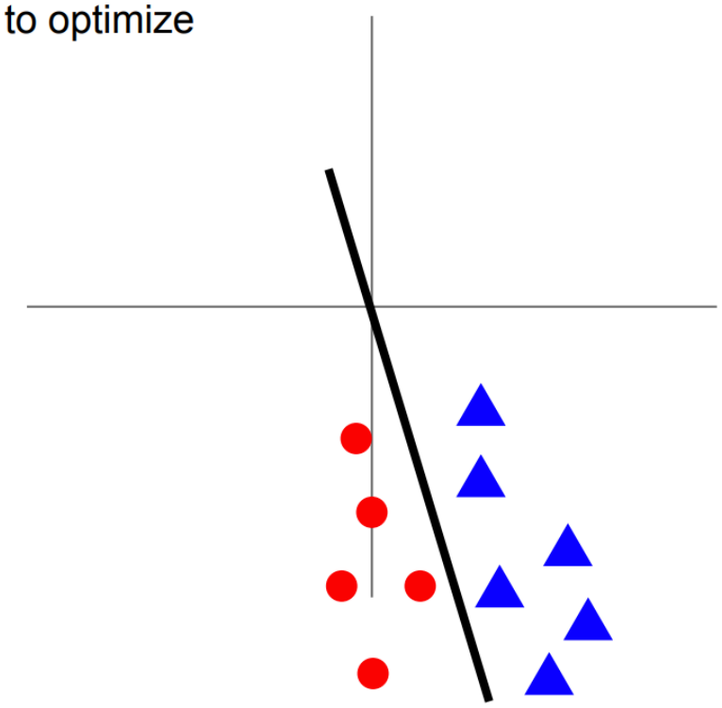

```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

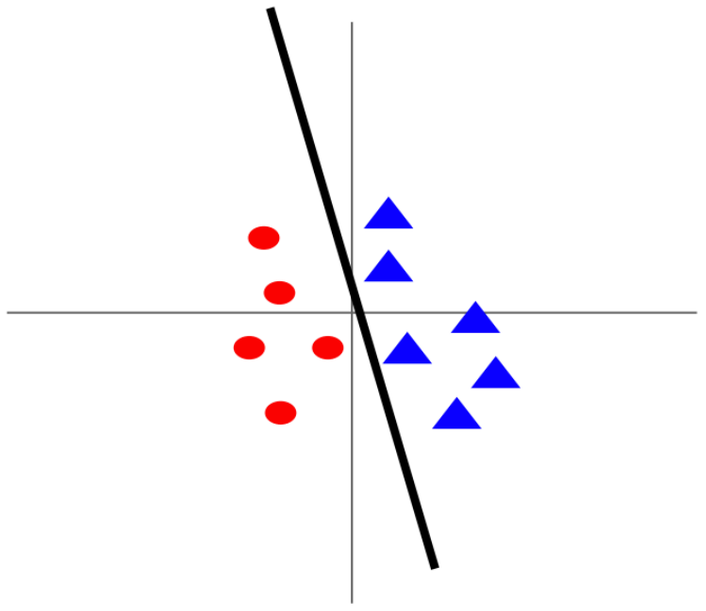
Normalization



Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

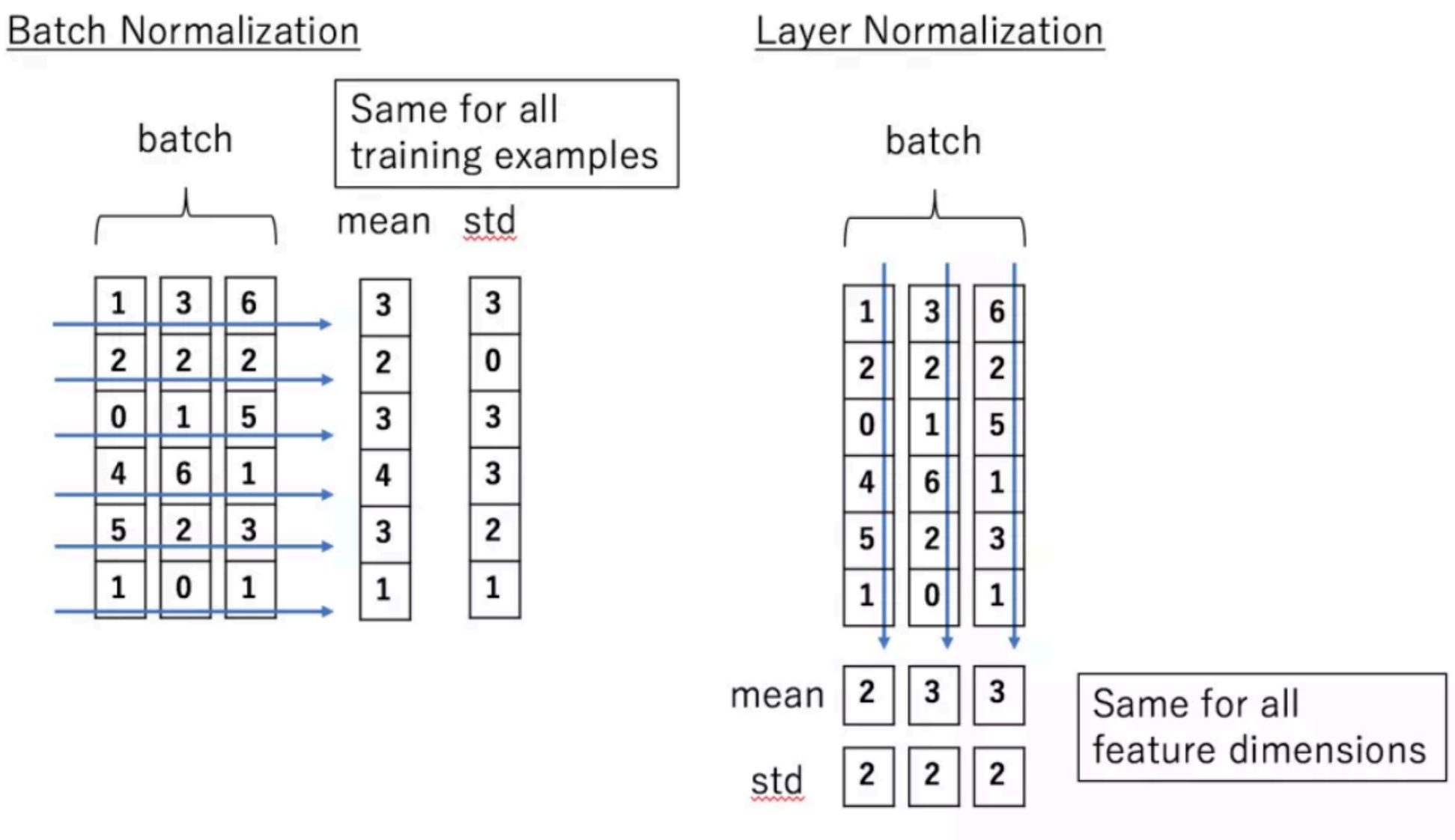
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

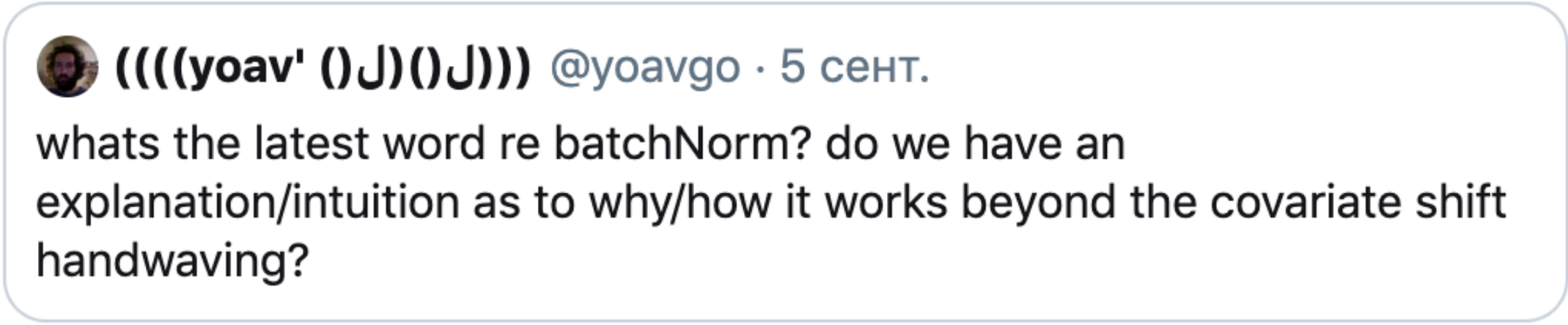
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Learnable

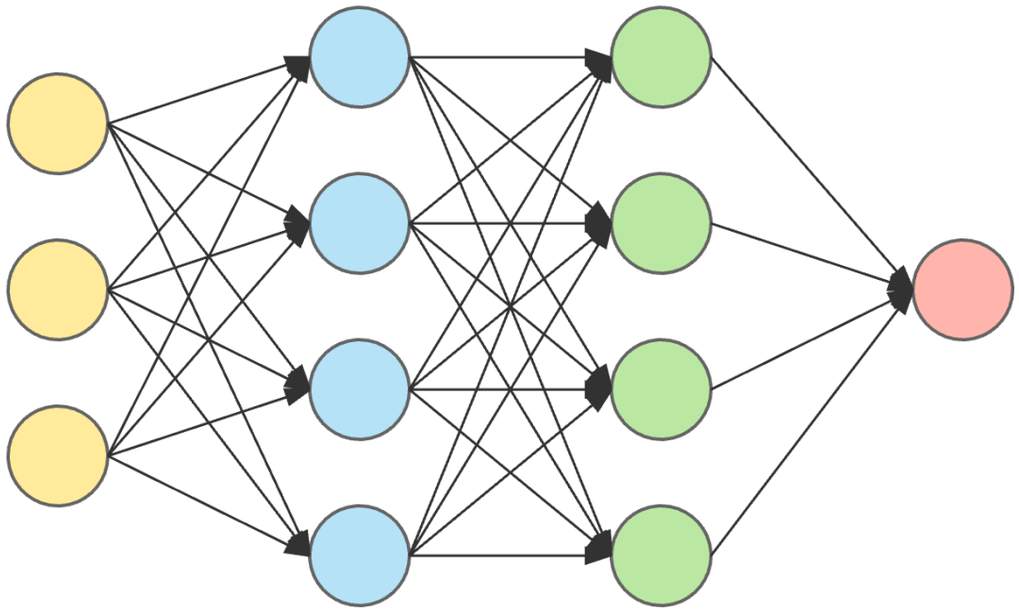
Batch Normalization



Batch Normalization



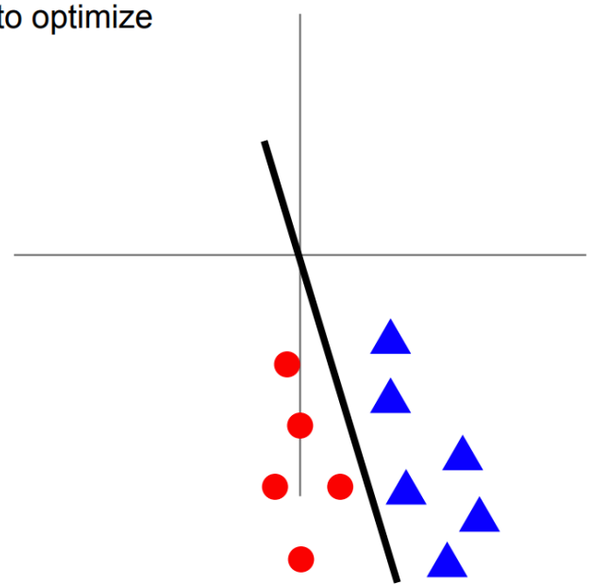
Batch Normalization



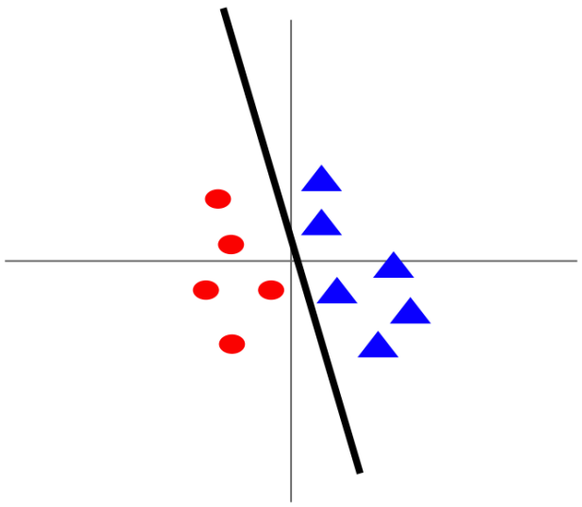
input layer hidden layer 1 hidden layer 2 output layer

Internal covariate shift

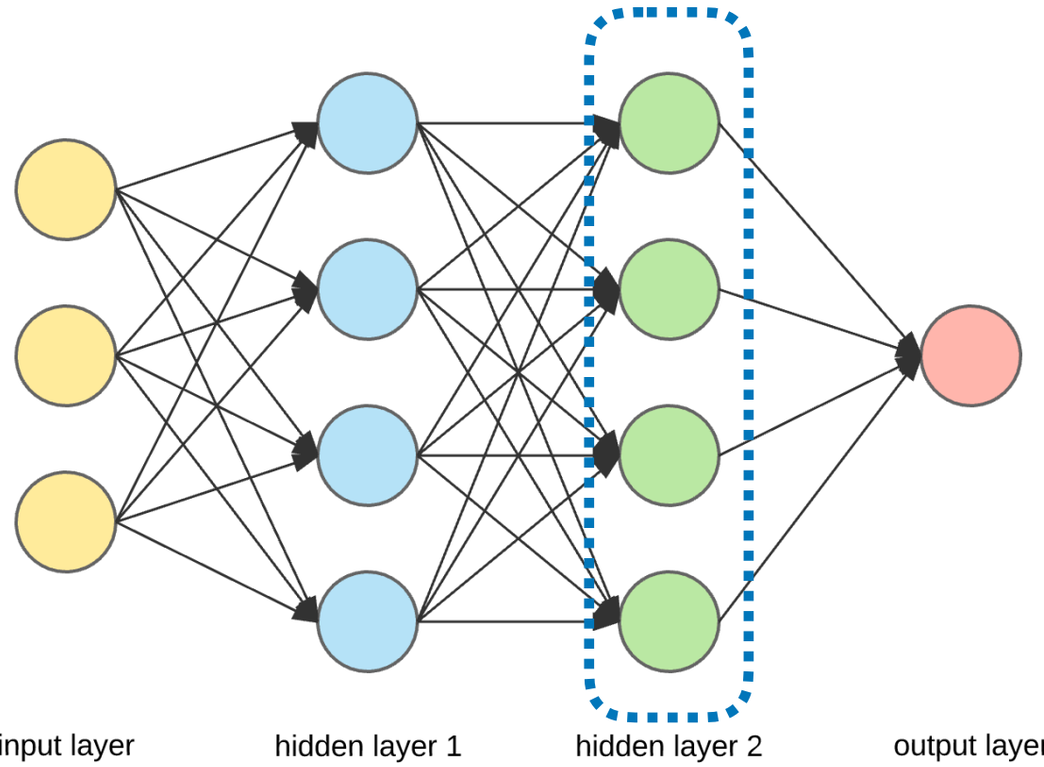
Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize

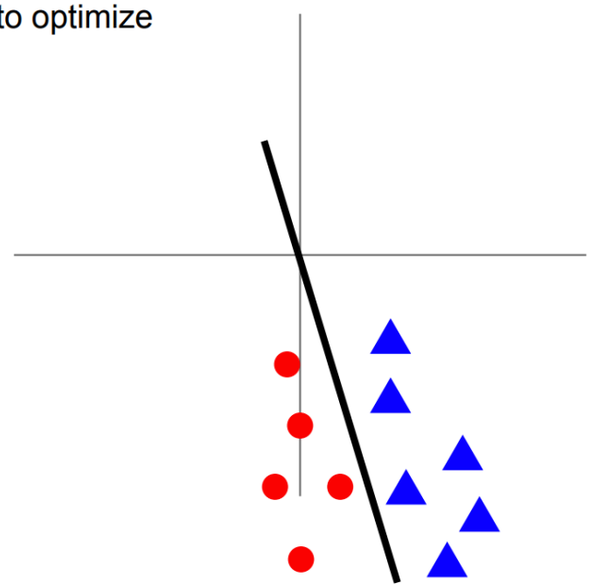


Batch Normalization

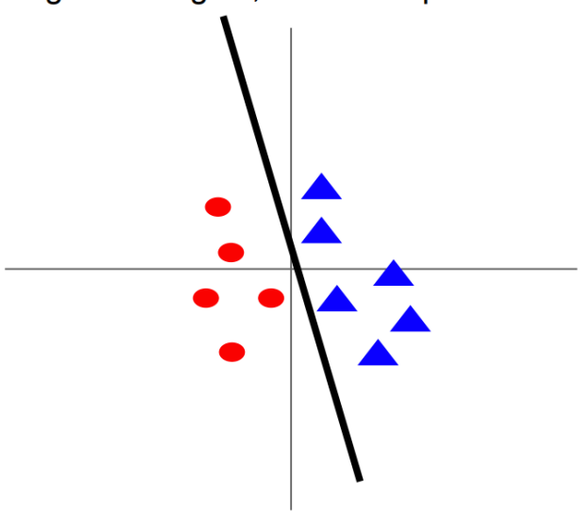


Internal covariate shift

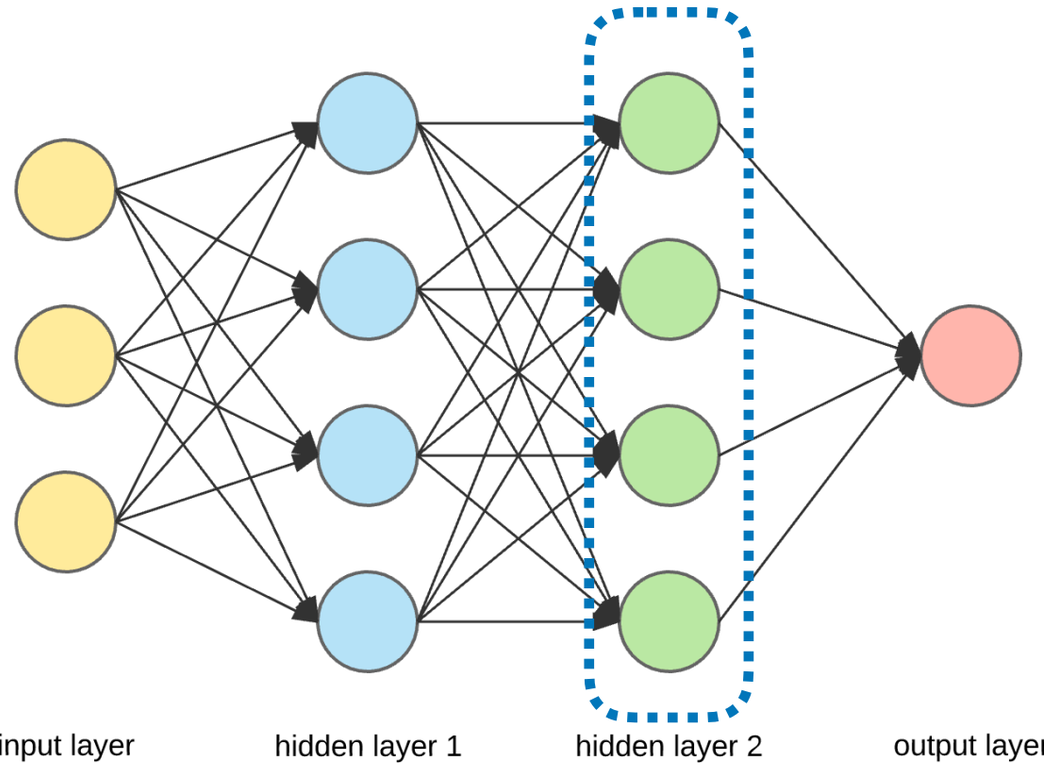
Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



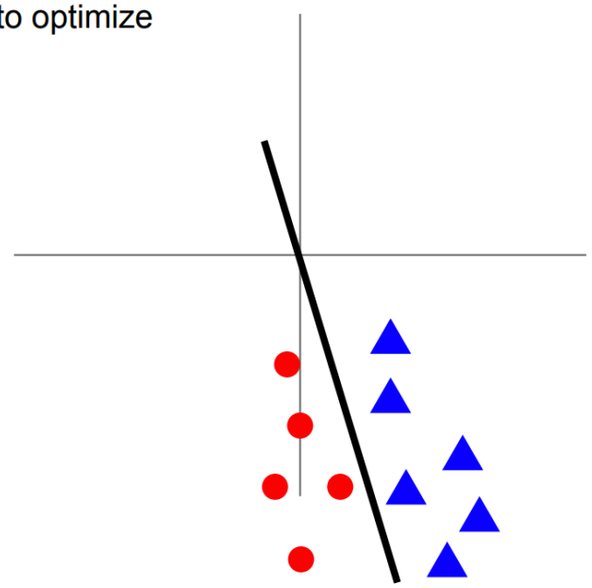
Batch Normalization



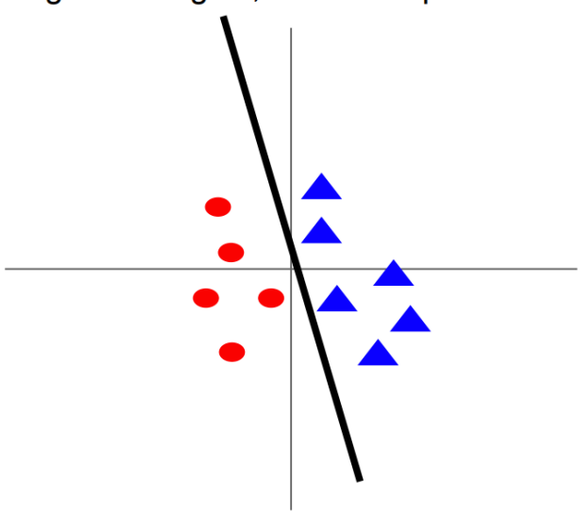
Internal covariate shift

During training
weights need
to continuously adapt
to the new distribution
of their inputs

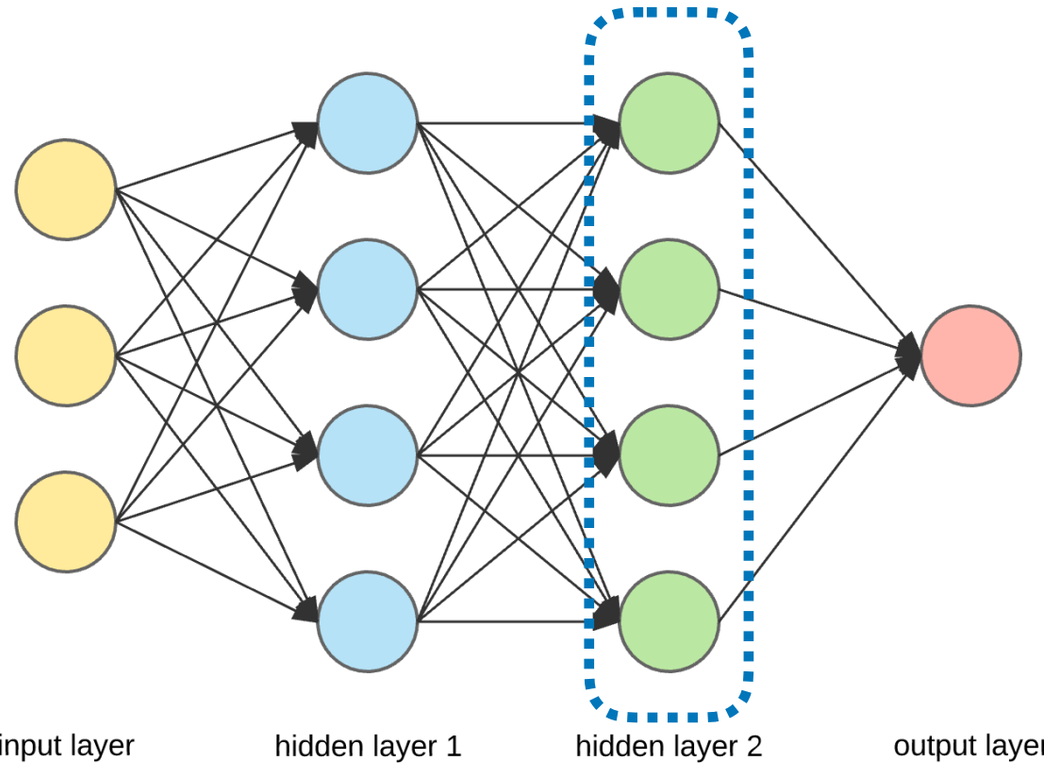
Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Batch Normalization

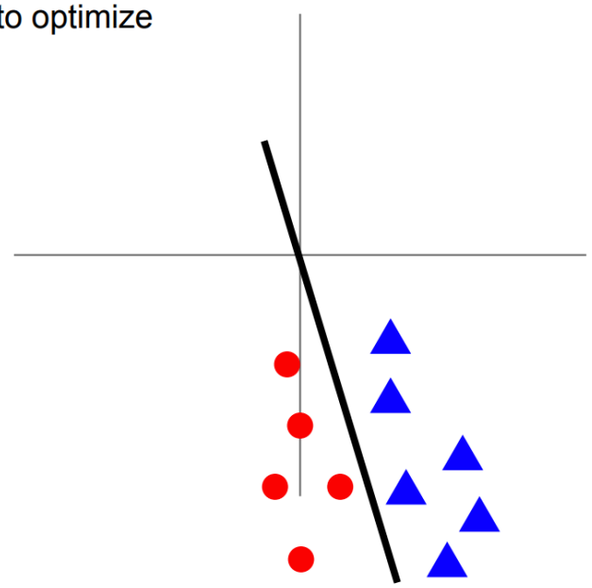


Internal covariate shift

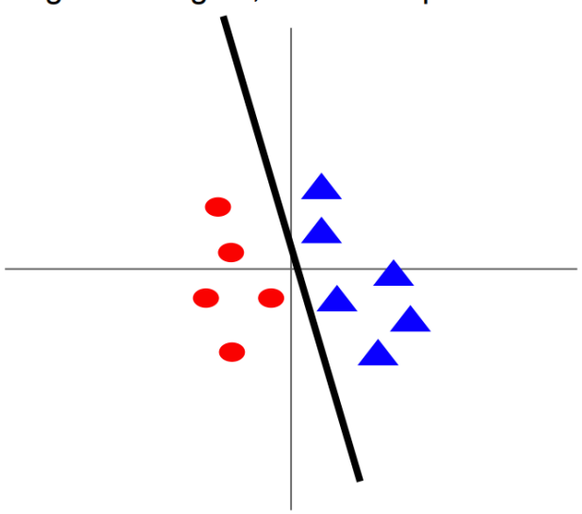
During training weights need to continuously adapt to the new distribution of their inputs

Vanishing/Exploding Gradients

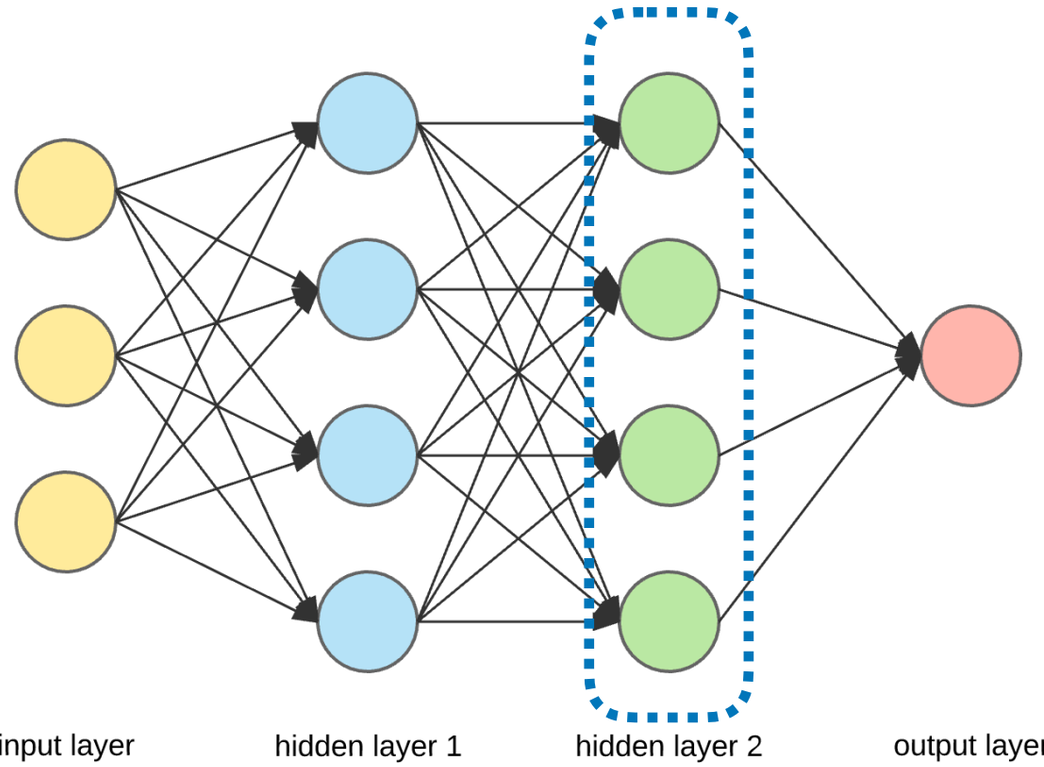
Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



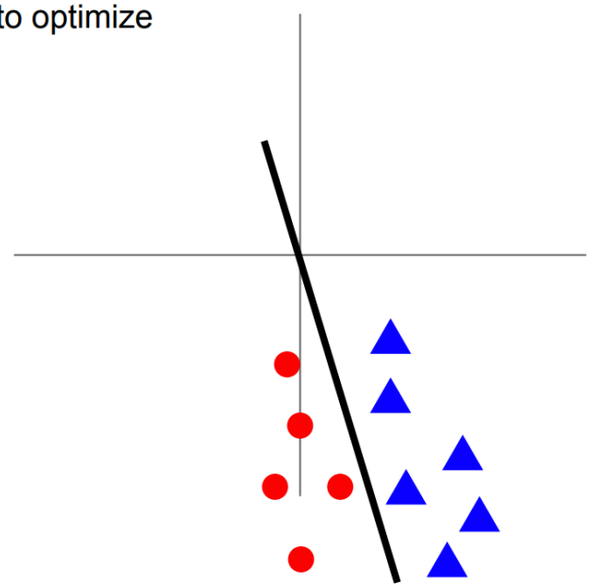
After normalization: less sensitive to small changes in weights; easier to optimize



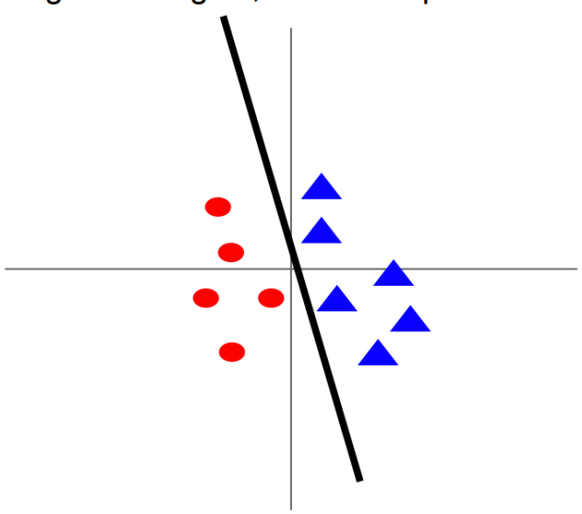
Batch Normalization



Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



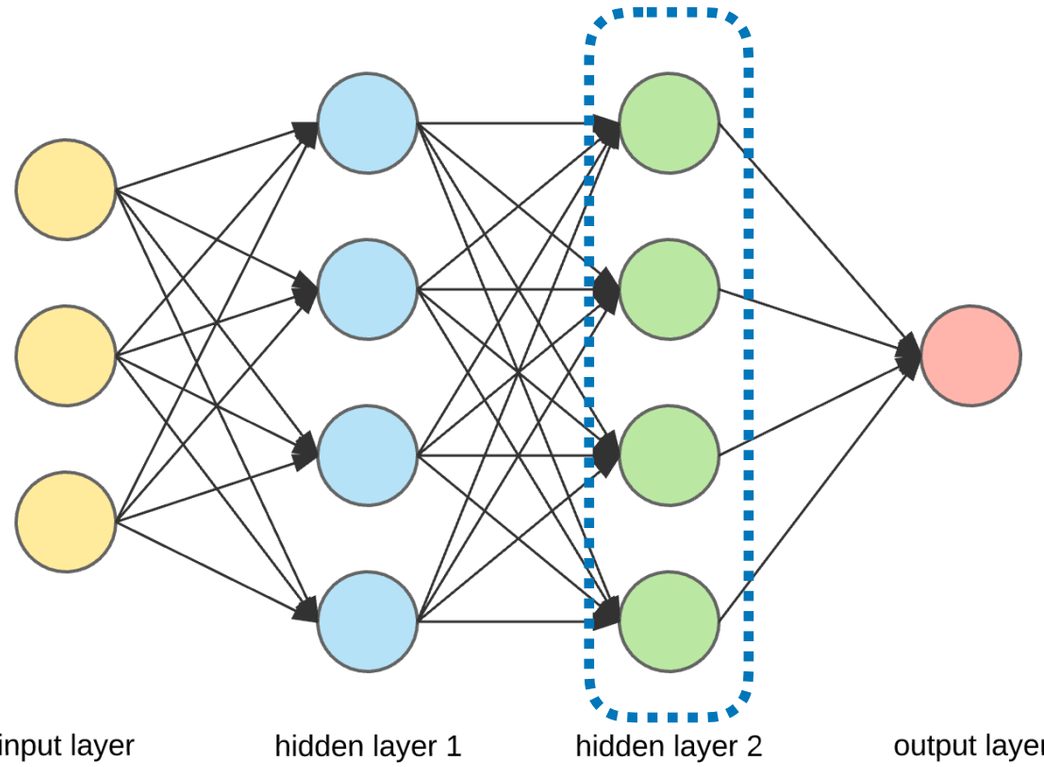
Internal covariate shift

During training weights need to continuously adapt to the new distribution of their inputs

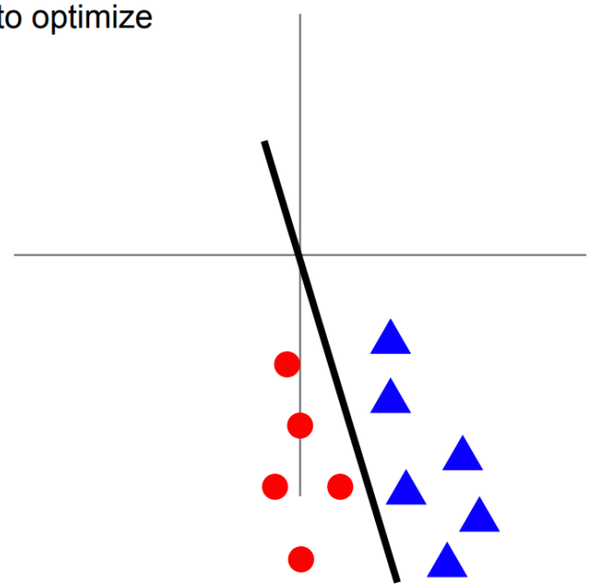
Vanishing/Exploding Gradients

Faster convergence

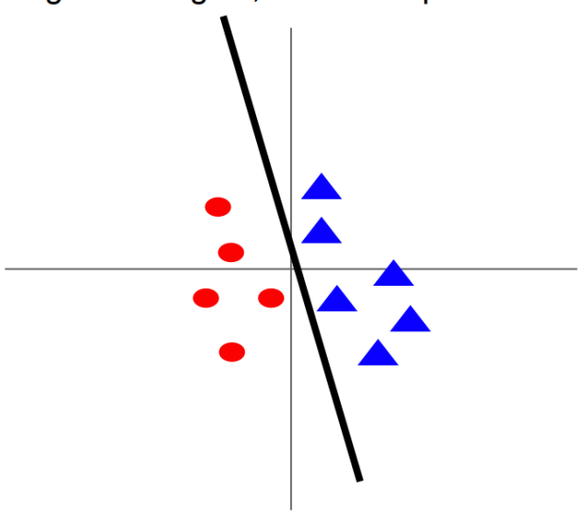
Batch Normalization



Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Internal covariate shift

During training weights need to continuously adapt to the new distribution of their inputs

Vanishing/Exploding Gradients

Faster convergence

Before activation

Batch Normalization

Во время предсказания батч-нормализация является линейным слоем:

$$\hat{x} = \frac{x - \mathbb{E}[x]}{\sqrt{\mathbb{D}[x] + \epsilon}}$$

$$y = \gamma \cdot \hat{x} + \beta$$

$$y = \frac{\gamma}{\sqrt{\mathbb{D}[x] + \epsilon}} \cdot x + (\beta - \frac{\gamma \mathbb{E}[x]}{\sqrt{\mathbb{D}[x] + \epsilon}})$$

$\mathbb{E}[x]$ и $\mathbb{D}[x]$ вычисляются по всему обучающему множеству. На практике статистики вычисляются во время обучения экспоненциальным средним: $E_{i+1} = (1 - \alpha)E_i + \alpha E_{\mathcal{B}}$

Training Tips

Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at loss curves

Thanks for your Attention!

Boris Zubarev

@bobazooba