11210IPT553000 Deep Learning in Biomedical Optical Imaging

Week6

Improving Deep Neural Networks

Hyperparameter Tuning, Regularization and Optimization

Instructor: Hung-Wen Chen @NTHU, Fall 2023

2023/10/16

SPONSORED BY

coursera

deeplearning.ai

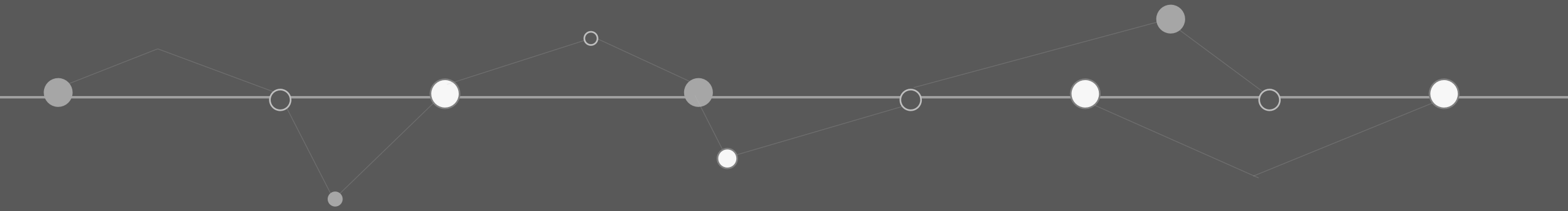
NVIDIA DEEP LEARNING INSTITUTE

NVIDIA DEEP LEARNING INSTITUTE CERTIFIED INSTRUCTOR

Outline of Today's Lecture

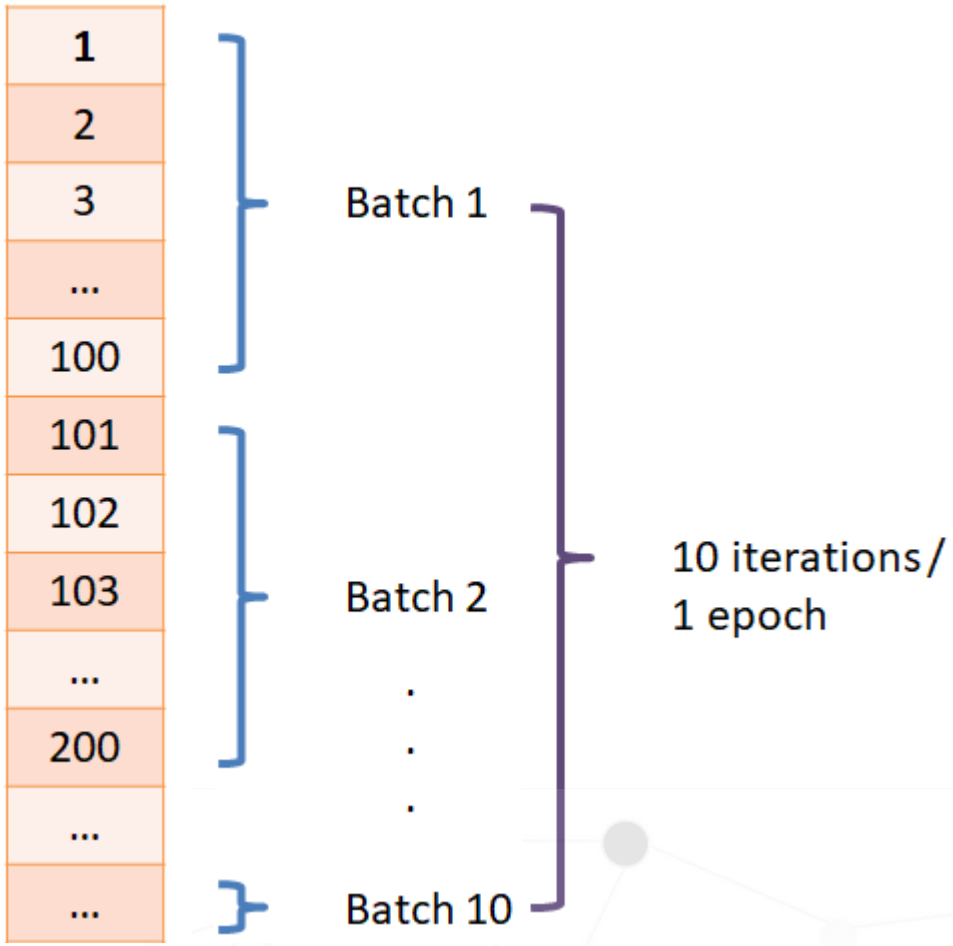
- Optimization algorithms (Course 2 Week 2)
- Hyperparameter Tuning (Course 2 Week 3)
- ML strategy (Course 3)
- Lab Practice: Build a CNN

Mini-Batch Gradient Descent



Mini-Batch Gradient Descent

Epoch / Batch Size/ Iteration



One **epoch**: one forward and backward pass of all training data

Batch size: the number of training examples in one forward and backward pass

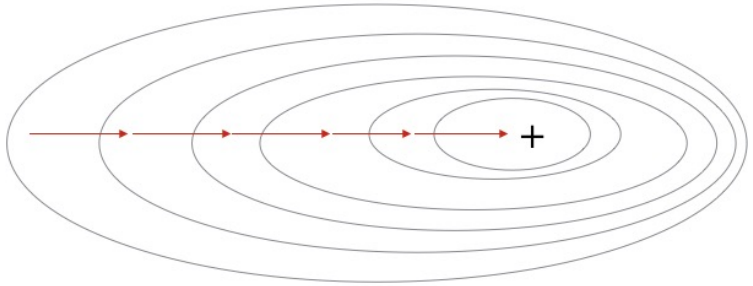
One **iteration**: number of passes

If we have 1,000 training data, and the **batch size** is 100. Then, we need 10 **iterations** to complete 1 **epoch**.

Mini-Batch Gradient Descent

Different gradient descents

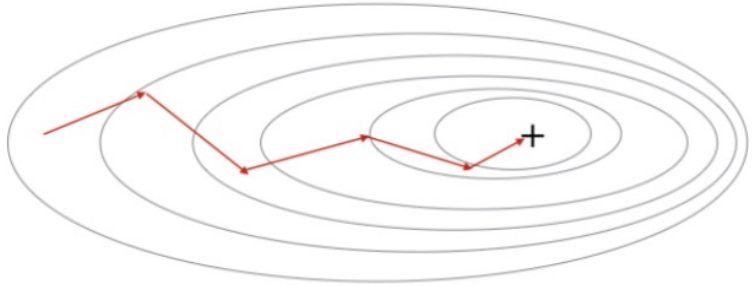
(batch size = M)



Batch GD
- Slowest
- Perfect gradient

Used with small data set (<2000)

(batch size = m)

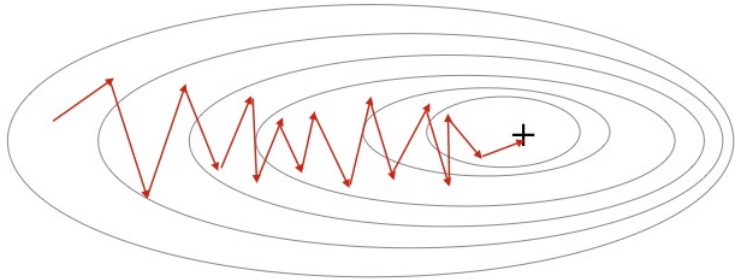


Mini-batch GD
- Compromise

Typical batch size (2^N)
64, 128, 256, 512

Make sure that your mini-batch fits in CPU/GPU memory.

(batch size = 1)



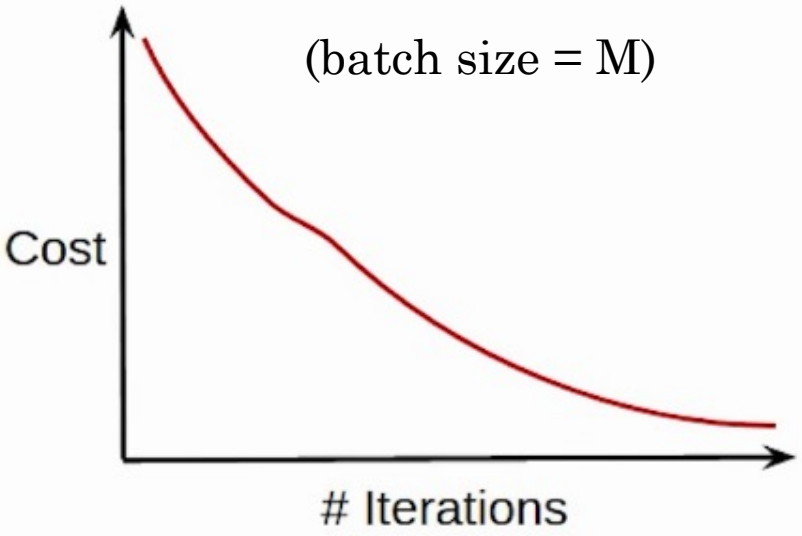
Stochastic GD
- Fastest
- Rough-estimate grad

Mini-Batch Gradient Descent

Training with mini-batch gradient descent

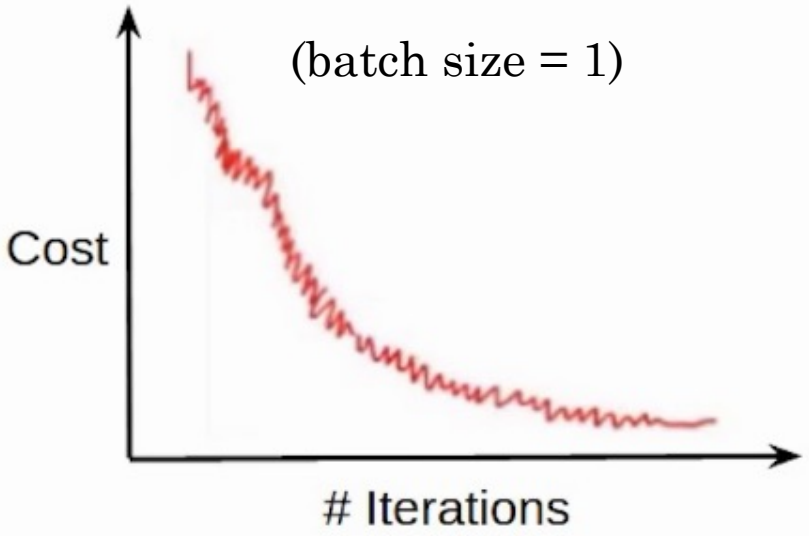
- Cost function reduces smoothly

(batch size = M)



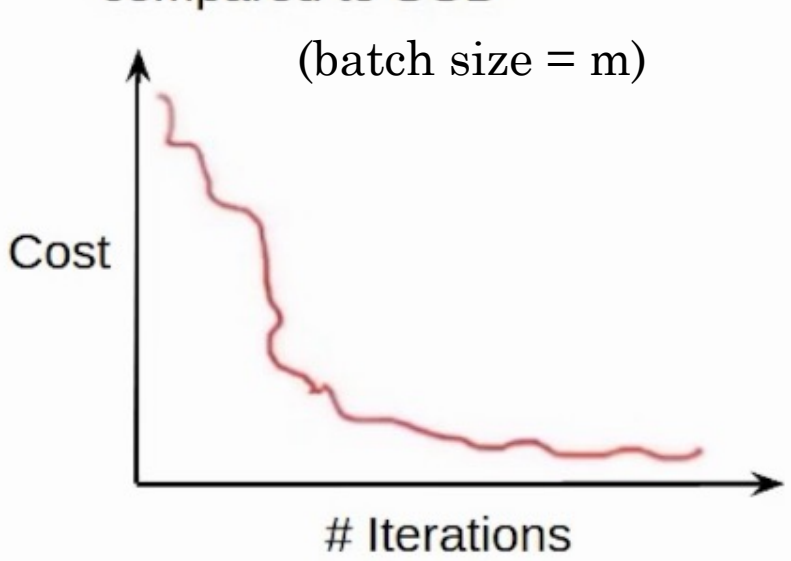
- Lot of variations in cost function

(batch size = 1)



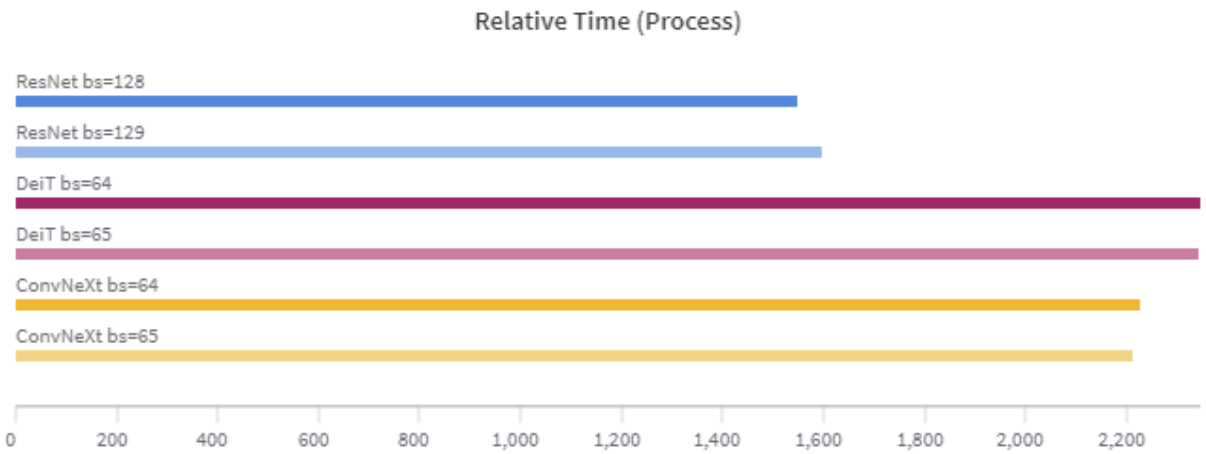
- Smoother cost function as compared to SGD

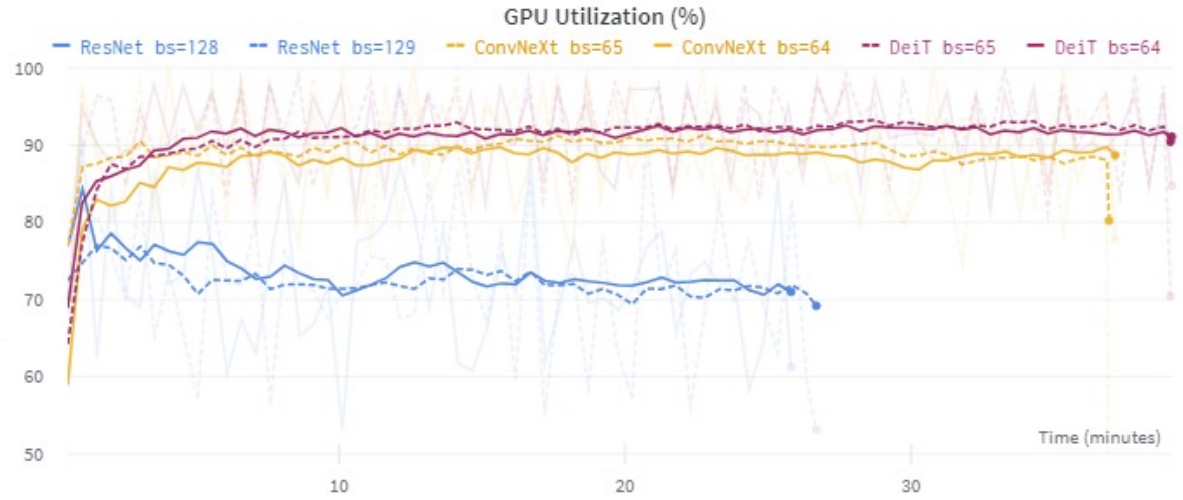
(batch size = m)

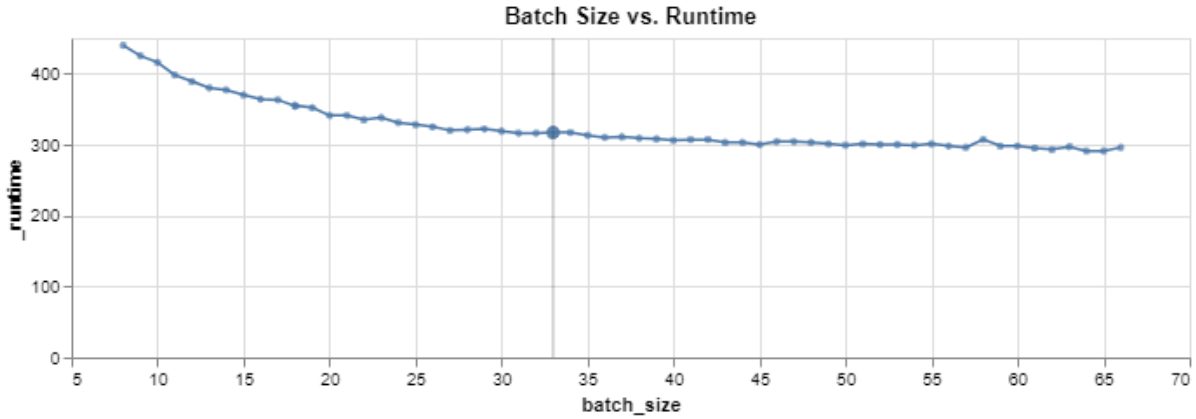


Mini-Batch Gradient Descent

Typical batch size (2^N)



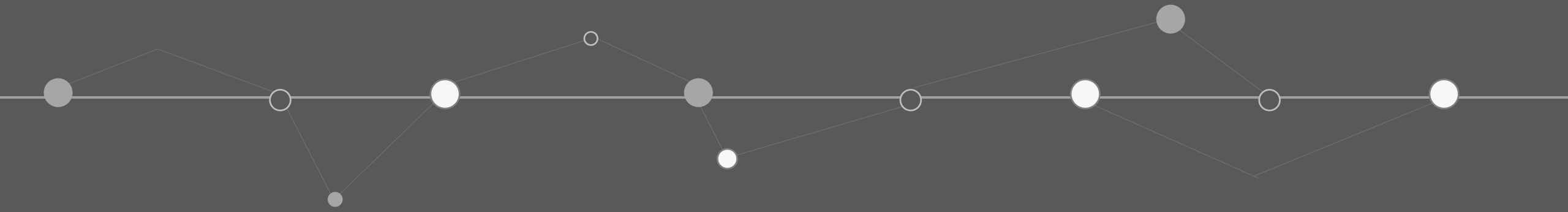




The chart shows the runtime of training a ConvNeXt for 5 epochs with different batch sizes.
Now, as expected, the runtime decreases when increasing the batch size until it hits a limit, but there are no sudden breaks at specific batch sizes. This suggests all the more that the restriction to powers of 2 is not necessary.

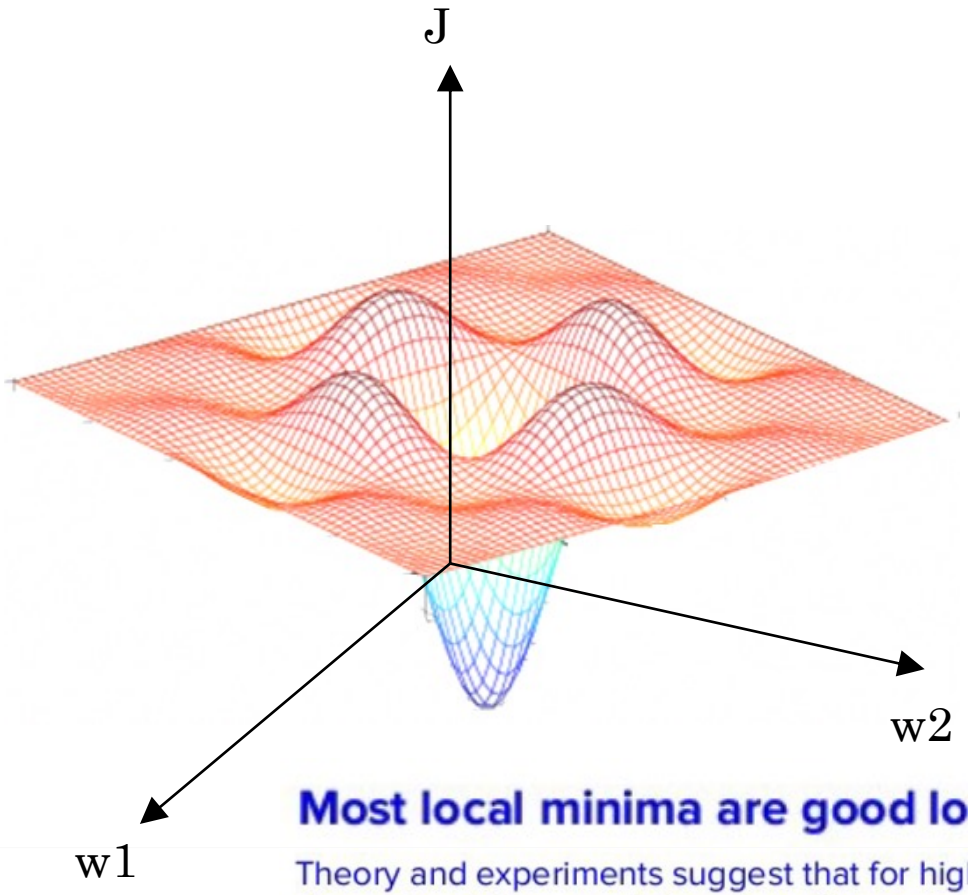
https://wandb.ai/datenzauberai/Batch-Size-Testing/reports/Do-Batch-Sizes-Actually-Need-to-be-Powers-of-2---VmlldzoyMDkwNDQx

Local Minimum



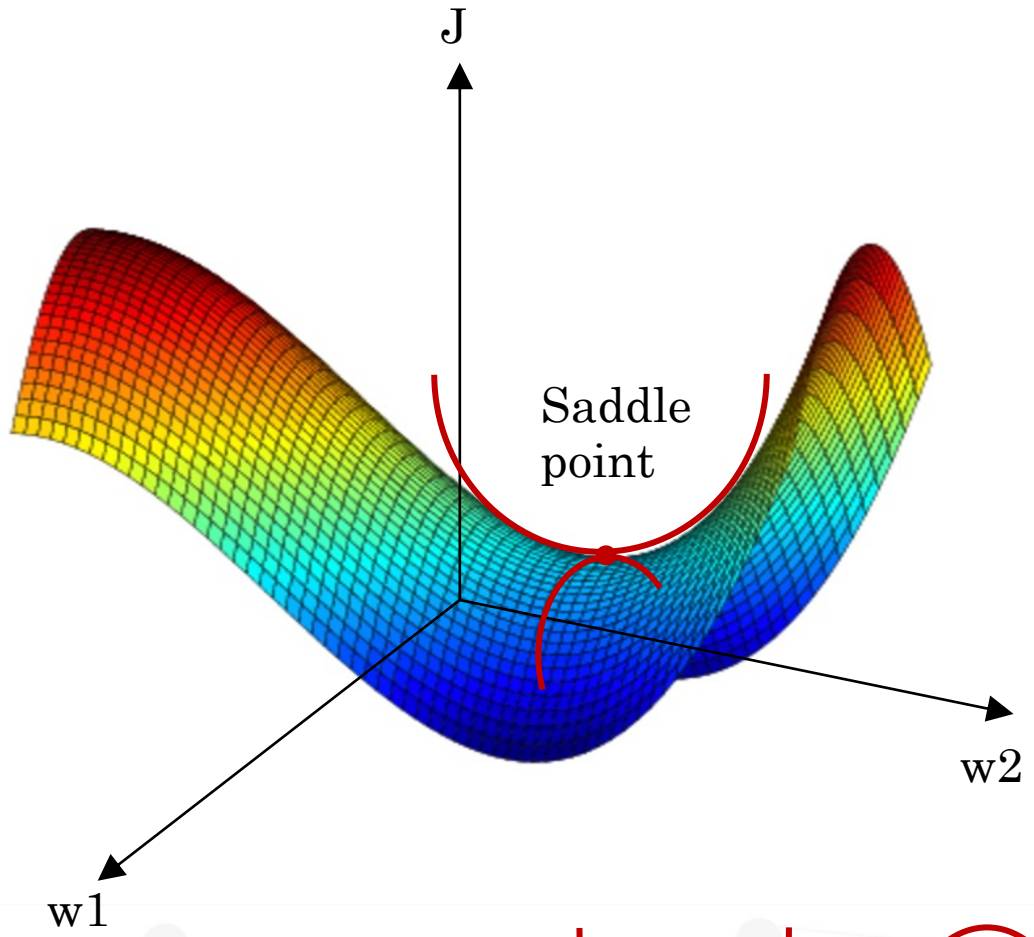
The Problem of Local Optima

Local optima in neural networks

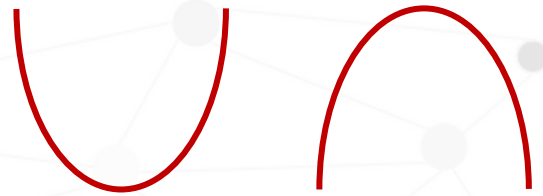


Most local minima are good local minima!

Theory and experiments suggest that for high dimensional deep models, value of loss function at most local minima is close to value of loss function at global minimum.



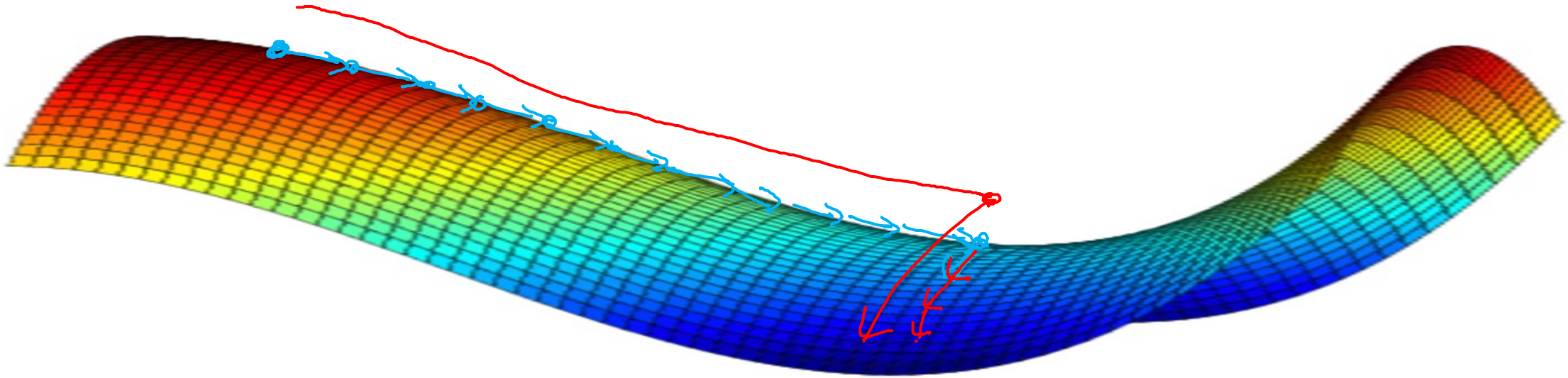
In high dimensional space
Gradient = 0



Choromanska et al. **The loss surfaces of multilayer networks**, AISTATS 2015 http://arxiv.org/abs/1412.0233

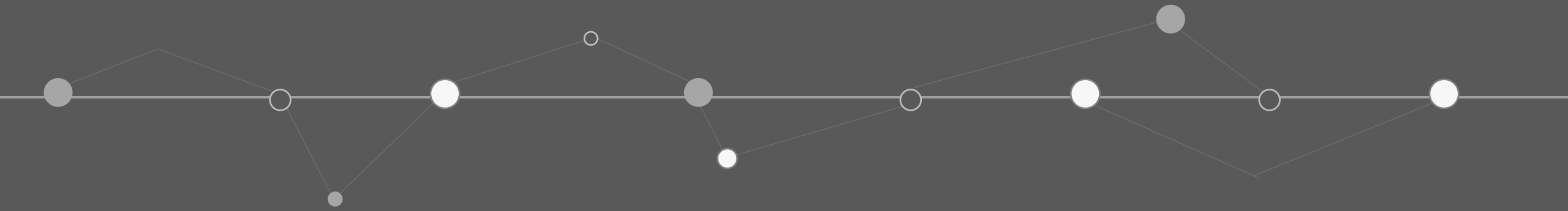
The problem of local optima

Problem of plateaus

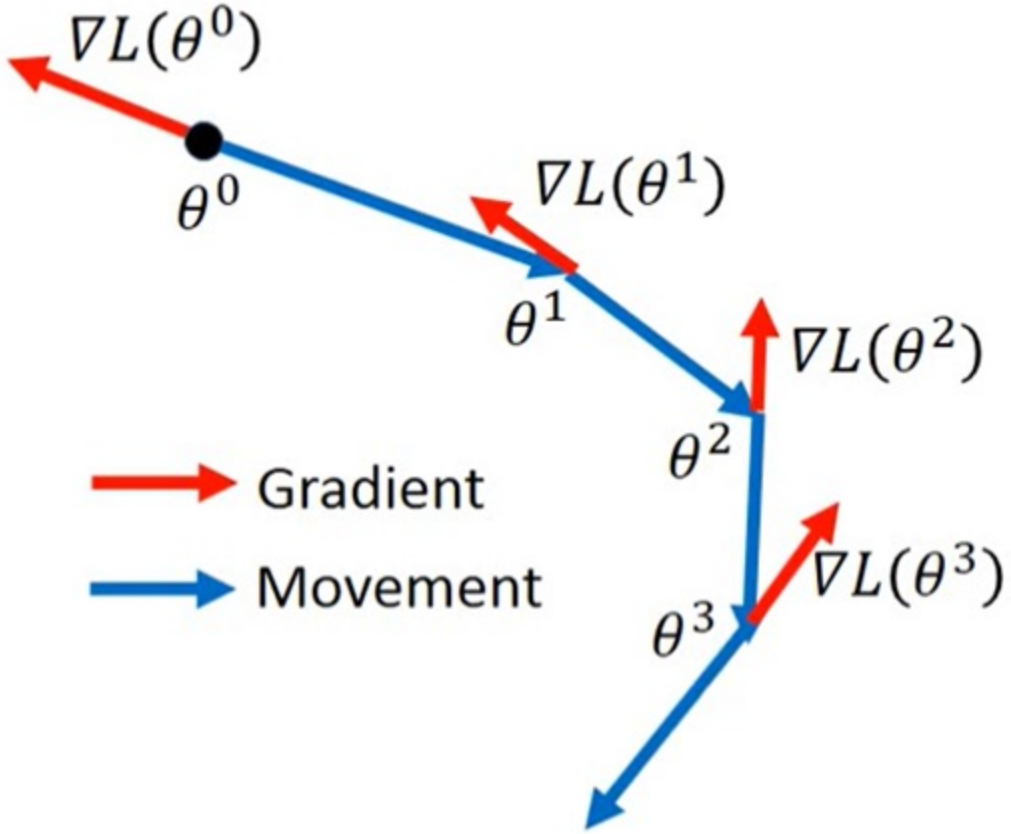


- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow

Momentum, RMSprop, and Adam



Gradient descent



Start at position θ^0

Compute gradient at θ^0

Move to $\theta^1 = \theta^0 - \eta\nabla L(\theta^0)$

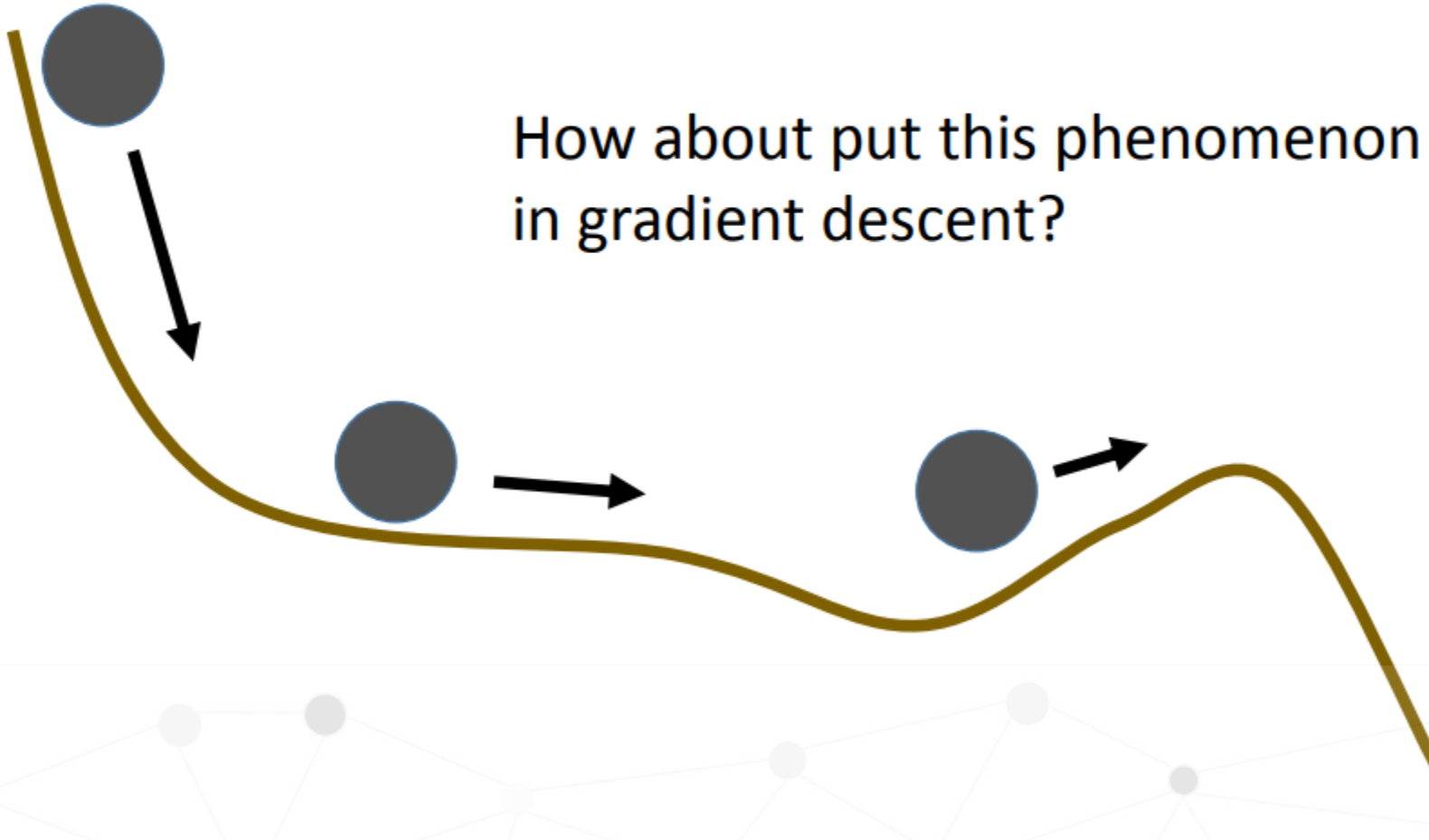
Compute gradient at θ^1

Move to $\theta^2 = \theta^1 - \eta\nabla L(\theta^1)$

⋮

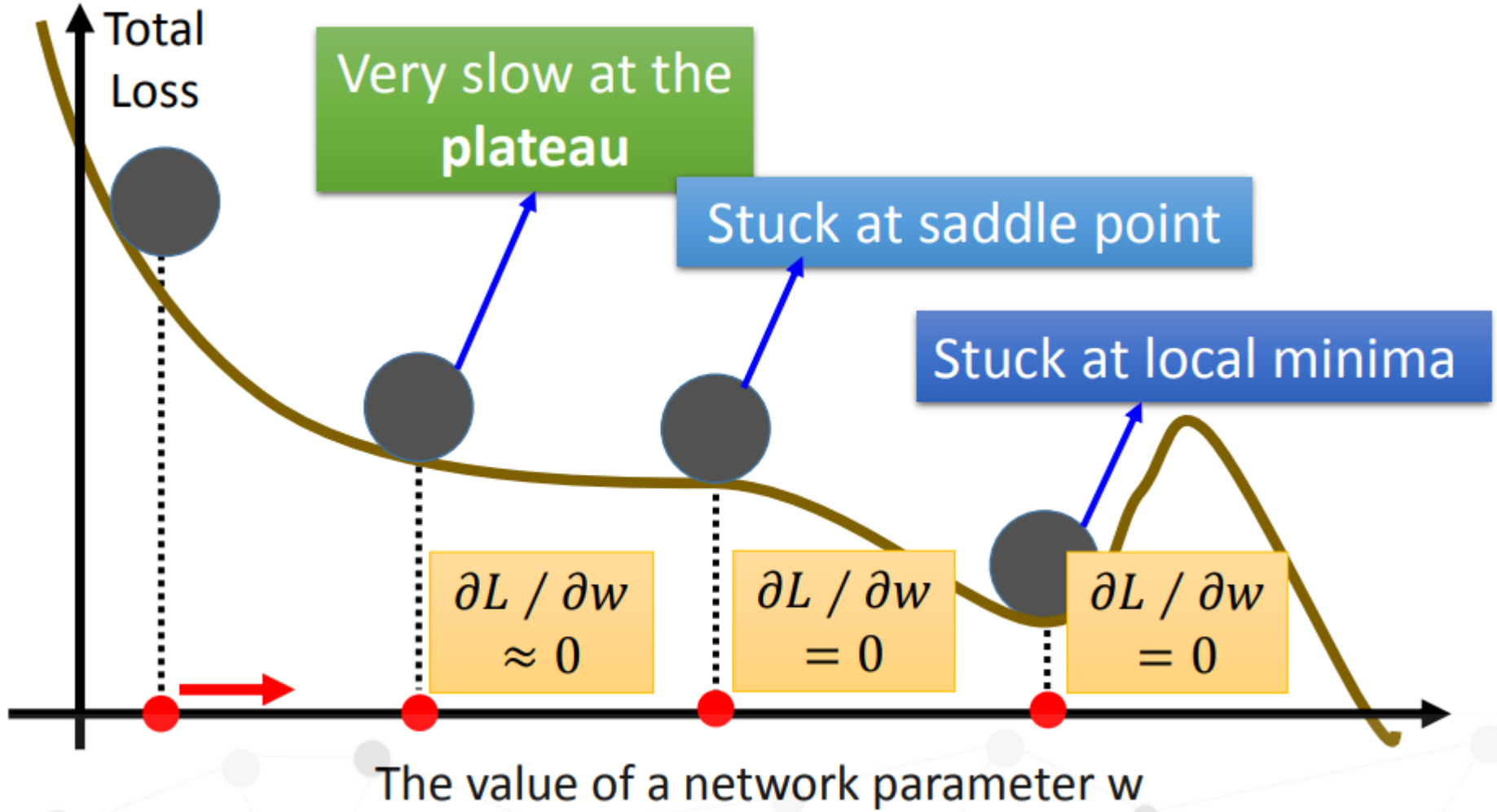
Stop until $\nabla L(\theta^t) \approx 0$

In physical world……



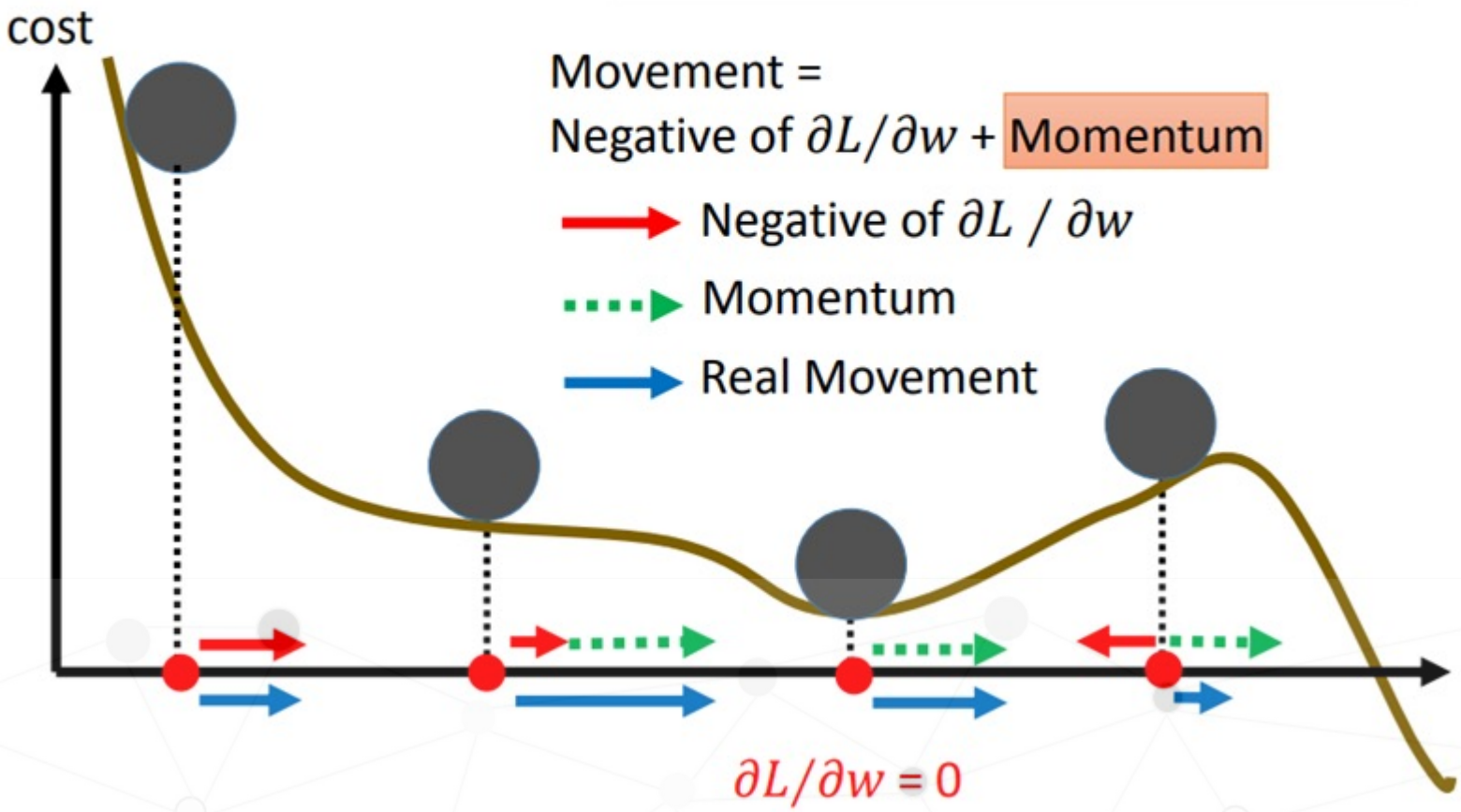
Mini-Batch Gradient Descent

Issues with gradient descent

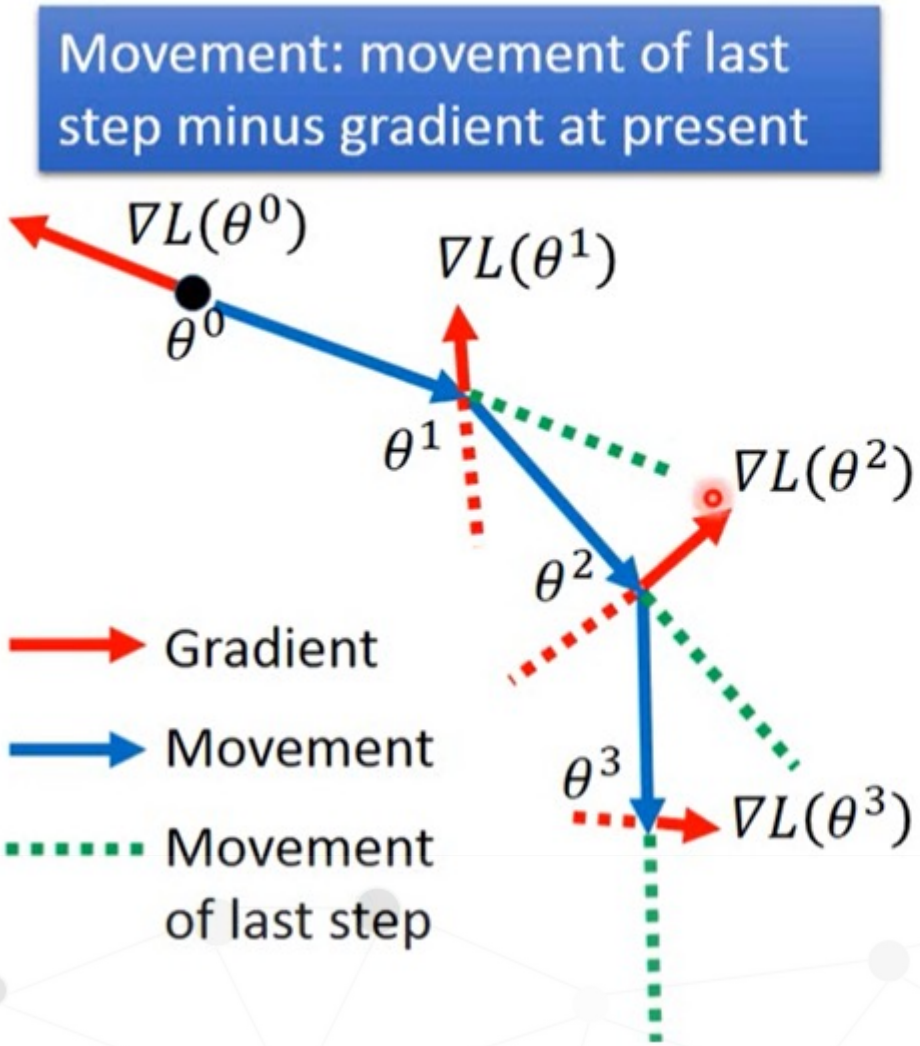


Momentum

Still not guarantee reaching global minima, but give some hope ……



Different gradient descents



Start at point θ^0

Movement $v^0=0$

Compute gradient at θ^0

Movement $v^1 = \lambda v^0 - \eta \nabla L(\theta^0)$

Move to $\theta^1 = \theta^0 + v^1$

Compute gradient at θ^1

Movement $v^2 = \lambda v^1 - \eta \nabla L(\theta^1)$

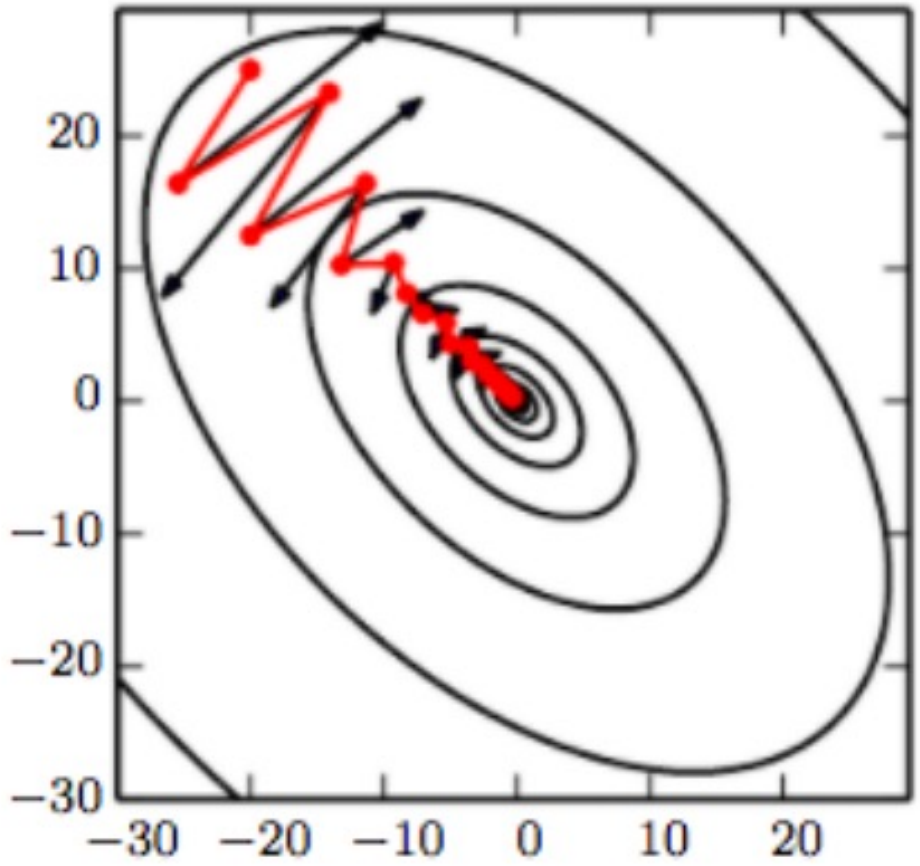
Move to $\theta^2 = \theta^1 + v^2$

Movement not just based on gradient, but previous movement.

Hung-yi Lee

Momentum

- In the high curvature directions, the gradients cancel each other out, so momentum dampens the oscillations.
- In the low curvature directions, the gradients point in the same direction, allowing the parameters to pick up speed.



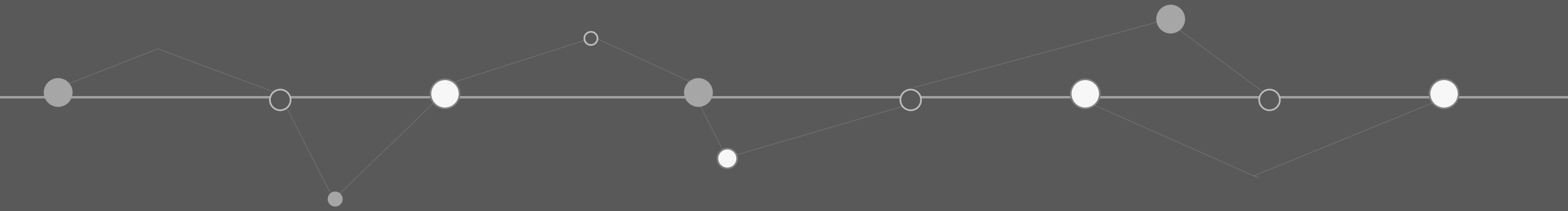
Adam optimization algorithm

Adaptive Moment Estimation
(Momentum + RMSprop)

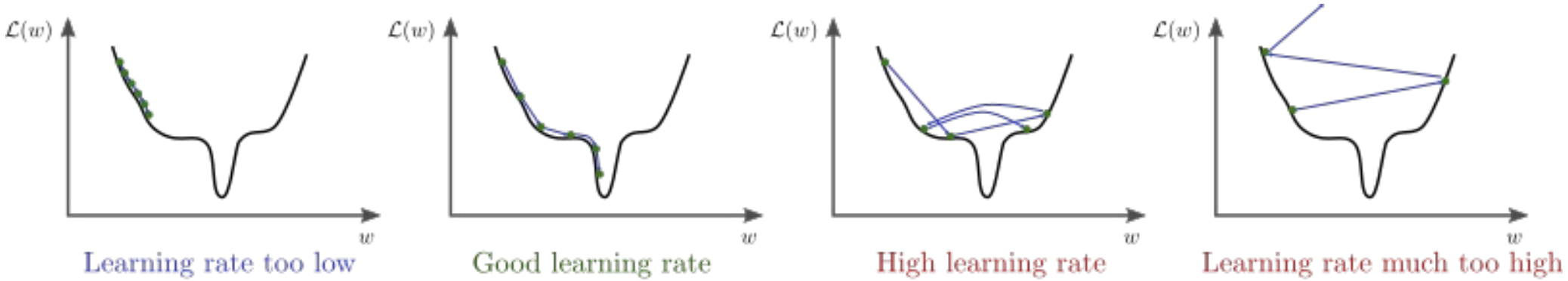
Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

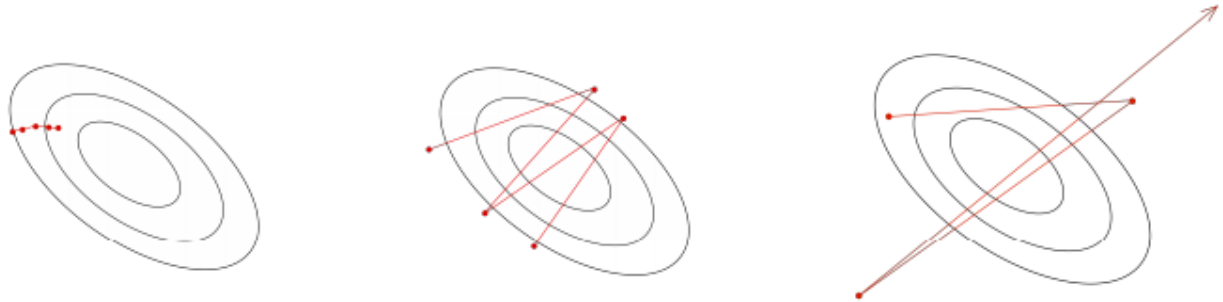
Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector) → for momentum
$v_0 \leftarrow 0$ (Initialize 2nd moment vector) → for RMSprop
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Learning Rate Decay



Learning rate settings



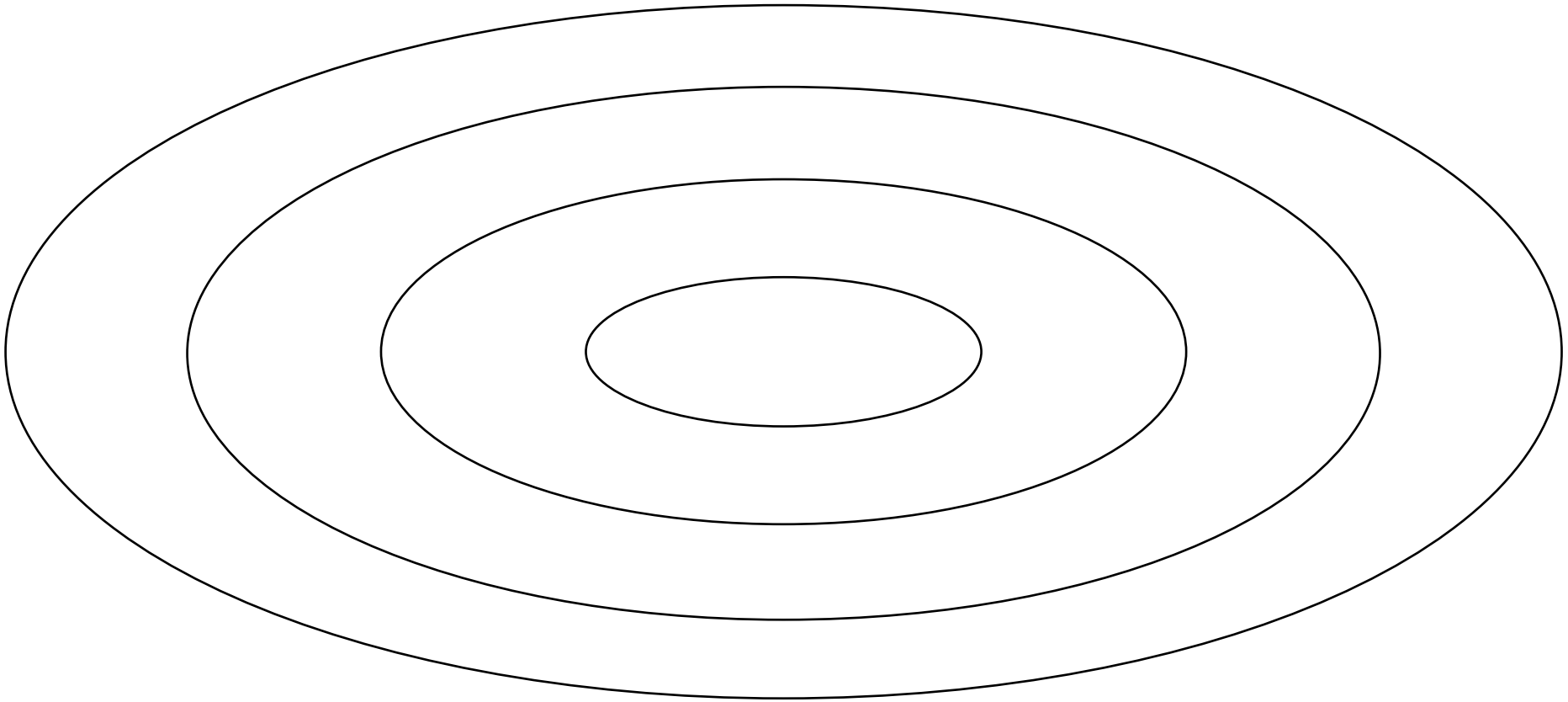


α too small: slow progress

α too large: oscillations

α much too large: instability

Adaptive learning rate

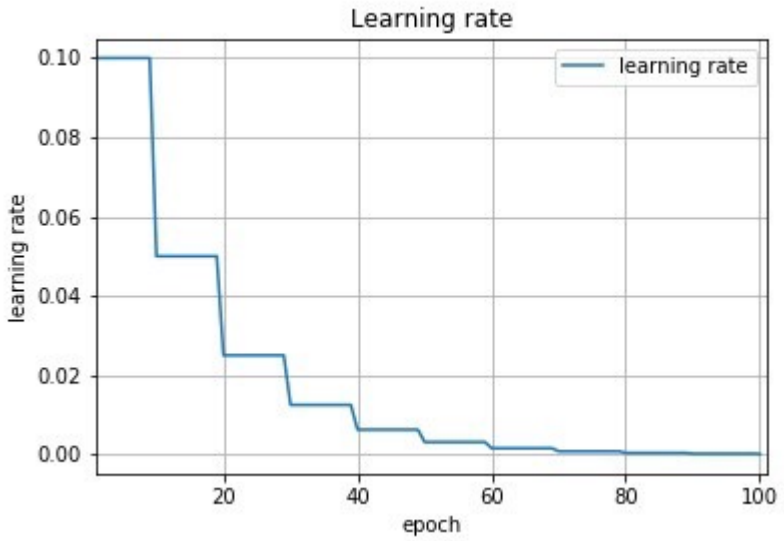


Reduce the learning rate by some factor every few epoch

- At the beginning, we are far from the destination, so we use larger learning rate
- After several epochs, we are close to the destination, so we reduce the learning rate

Learning rate decay

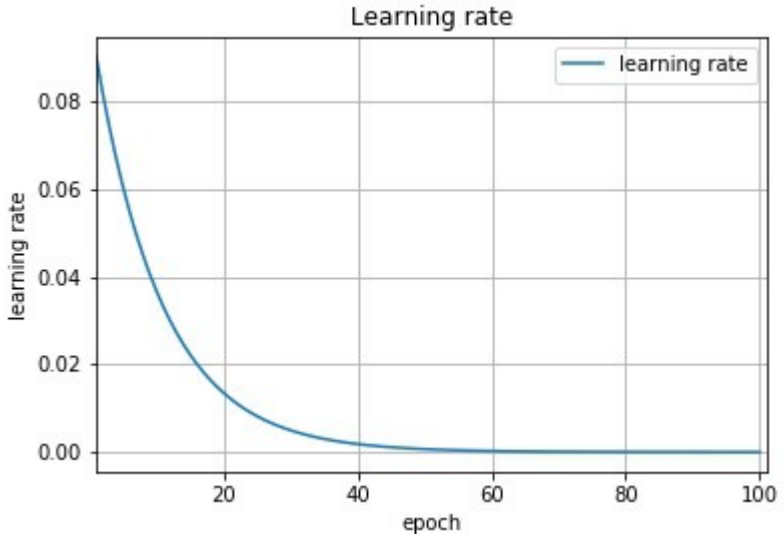
Step decay




```
def step_decay(epoch):
   initial_lrate = 0.1
   drop = 0.5
   epochs_drop = 10.0
   lrate = initial_lrate * math.pow(drop,
           math.floor((1+epoch)/epochs_drop))
   return lrate


lrate = LearningRateScheduler(step_decay)
```

Exponential decay




```
def exp_decay(epoch):
   initial_lrate = 0.1
   k = 0.1
   lrate = initial_lrate * exp(-k*t)
   return lrate


lrate = LearningRateScheduler(exp_decay)
```

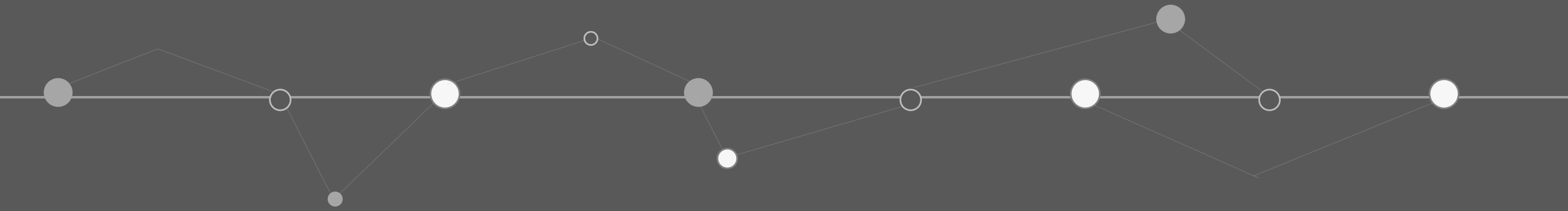
Keras

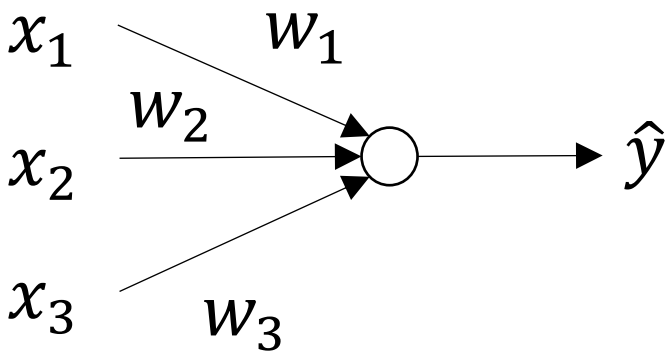
Learning rate schedules API

- ExponentialDecay
- PiecewiseConstantDecay
- PolynomialDecay
- InverseTimeDecay
- CosineDecay
- CosineDecayRestarts

https://keras.io/api/optimizers/learning_rate_schedules/

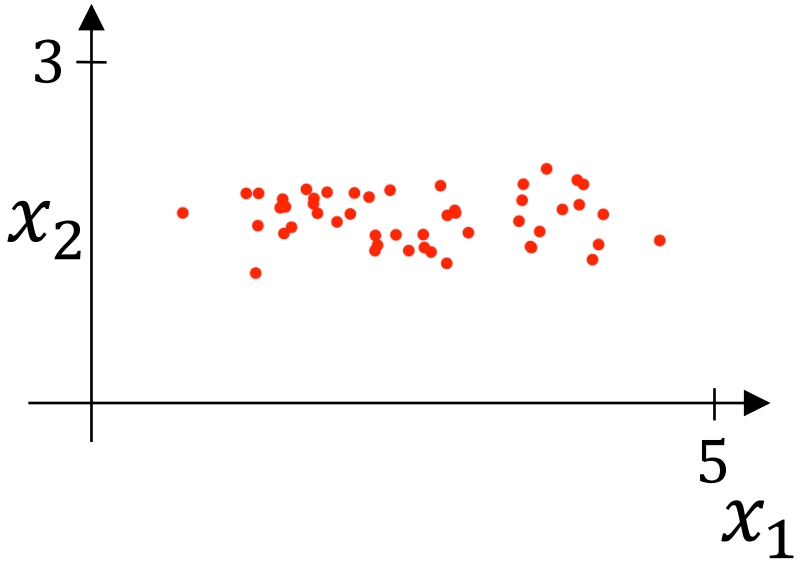
Batch Normalization



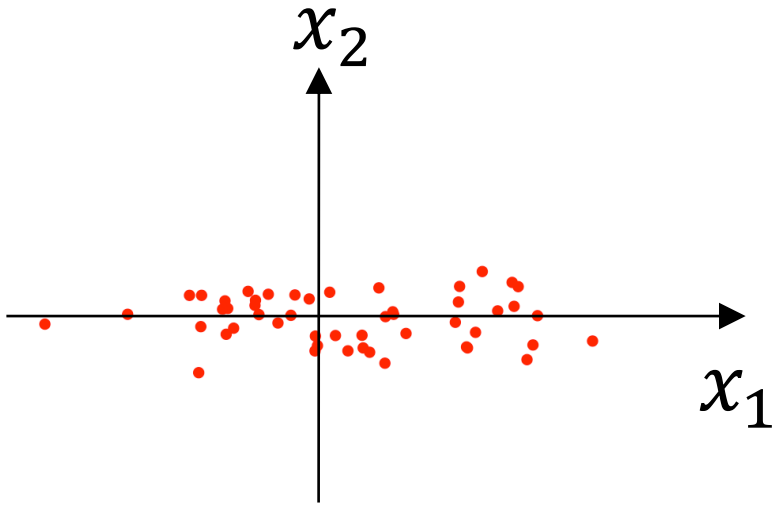


Normalizing Inputs

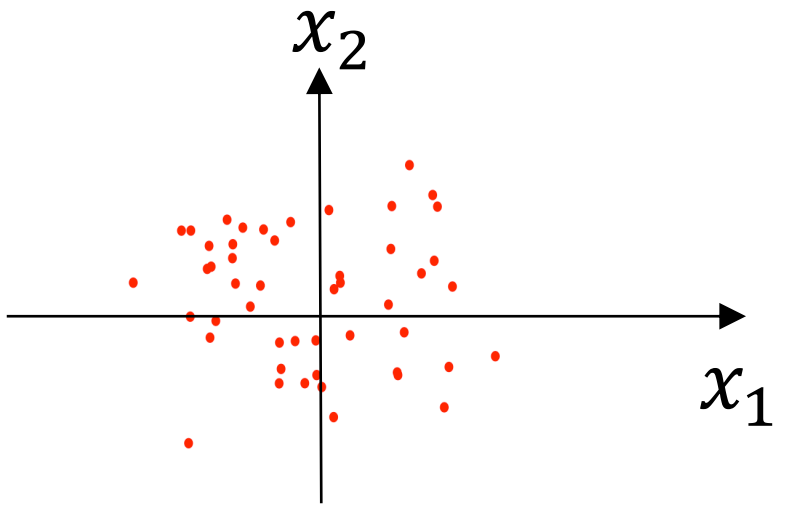
Normalizing training sets to speed up training



X



$X := X - \mu$



$$X := \frac{X - \mu}{\sigma^2}$$

$$\mu = \frac{1}{m}\sum_{i=1}^{m} X^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (X^{(i)})^2$$

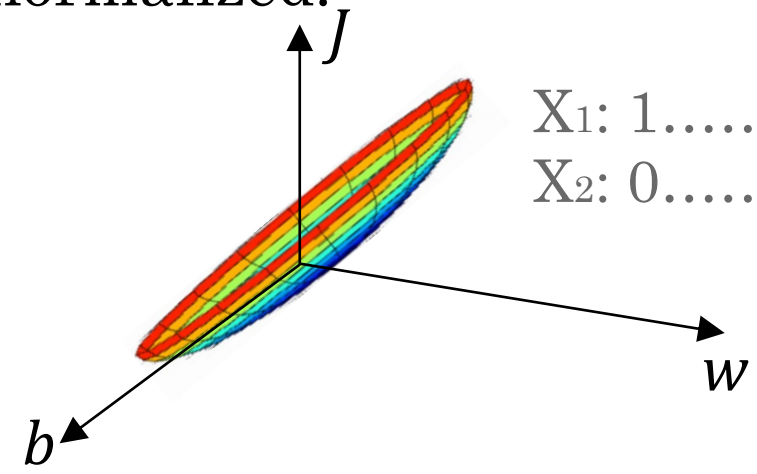
□ = 0
$\sigma^2 = 1$

Normalizing Inputs

Why normalize inputs?

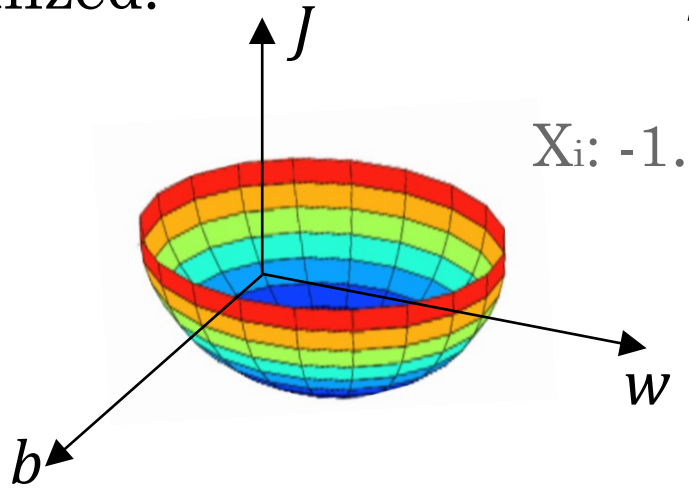
$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

Unnormalized:

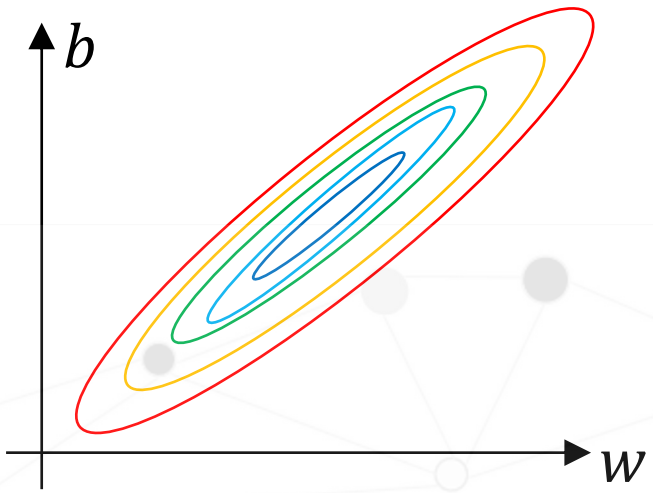


X_1: 1……1000
X_2: 0……0.01

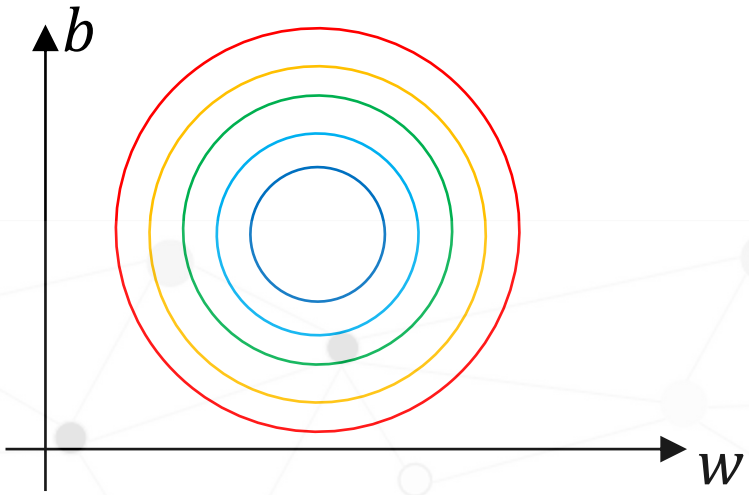
Normalized:



X_i: -1……1

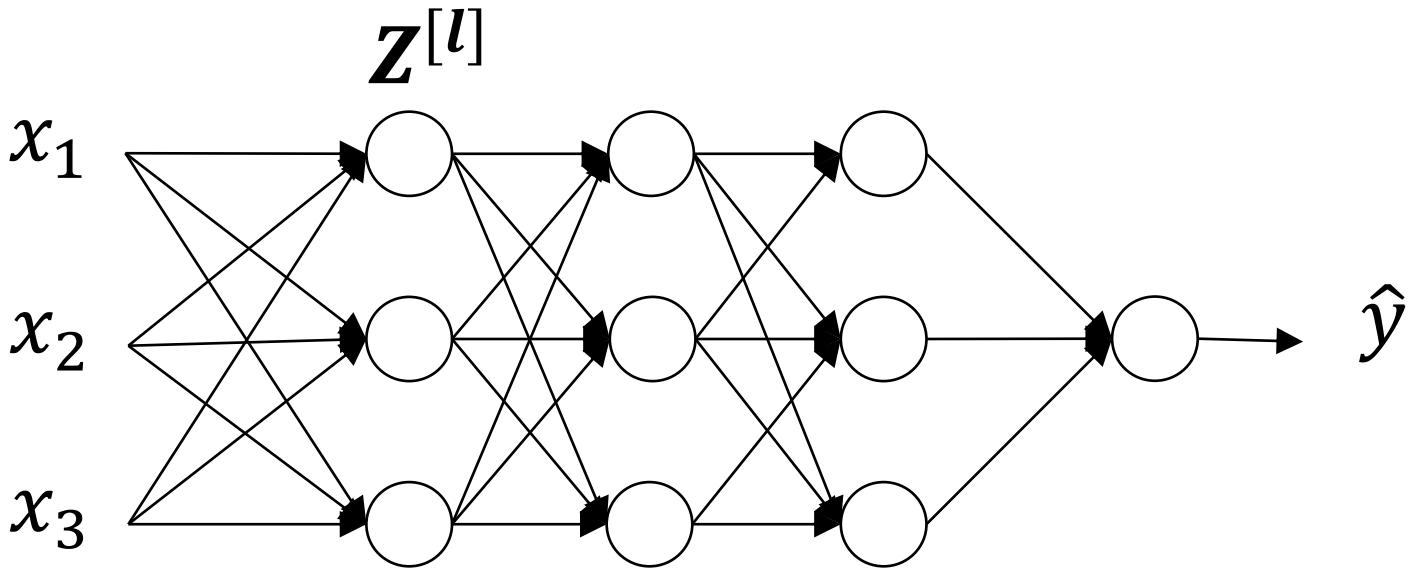


X_1: 0……1
X_2: -1……1
X_3: 1……2



Batch Normalization

Adding batch norm to a network



Given some intermediate values in $z^{[l]}$: $z^{[l](1)}, z^{[l](2)}$, ……

$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z^{(i)} - \mu)^2$$

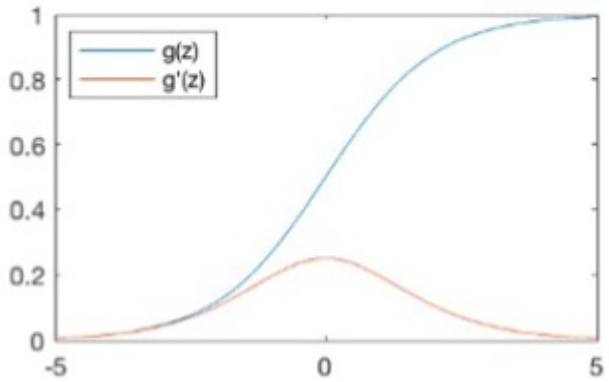
$$z^{(i)}_{\text{norm}} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

Normalizing Activations in a Network

Implementing batch norm

$$z^{(i)}_{\text{norm}} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

Sigmoid Function



Keras

BatchNormalization layer

BatchNormalization class

Use $\tilde{z}^{(i)}$ instead of $z^{(i)}_{\text{norm}}$

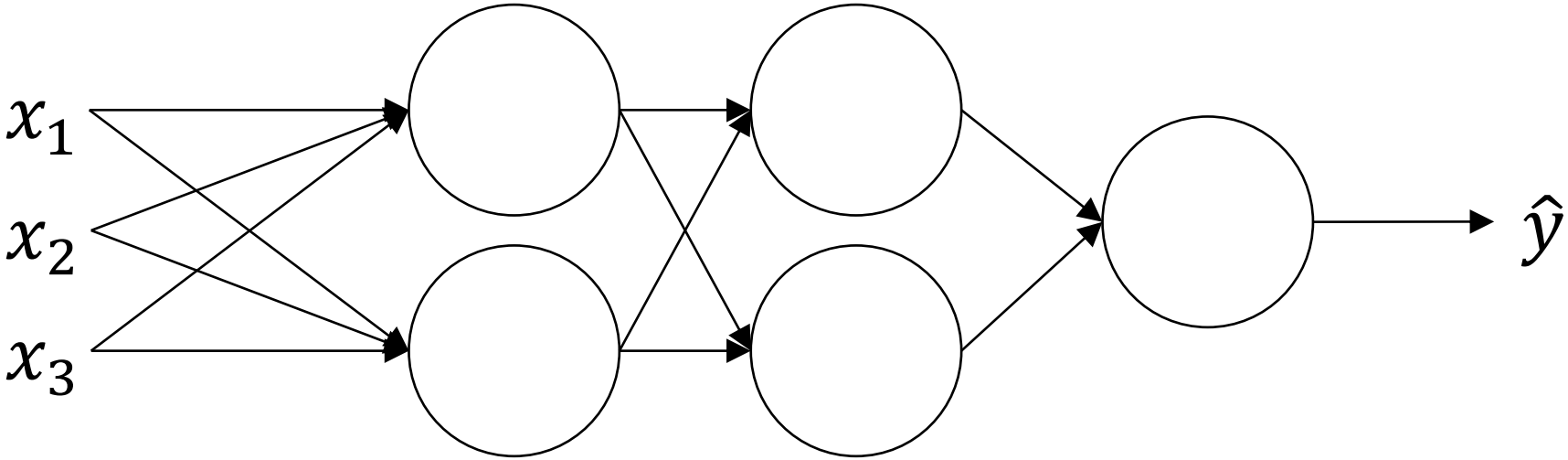
$$\tilde{z}^{(i)} = \gamma z^{(i)}_{\text{norm}} + \beta$$

γ and β are learnable parameters

https://keras.io/api/layers/normalization_layers/batch_normalization/

Fitting Batch Norm into a Neural Network

Adding batch norm to a network



$x^{[0]}$ → $W^{[1]}, b^{[1]}$ → $z^{[1]}$ → $a^{[1]} = g^{[1]}(z^{[1]})$ → $W^{[2]}, b^{[2]}$ → $z^{[2]}$ → $a^{[2]}$ ……

$x^{[0]}$ → $W^{[1]}, b^{[1]}$ → $z^{[1]}$ → $\gamma^{[1]}, \beta^{[1]}$ → $\tilde{z}^{[1]}$ → $a^{[1]} = g^{[1]}(\tilde{z}^{[1]})$ → $W^{[2]}, b^{[2]}$ → $z^{[2]}$ → $\gamma^{[2]}, \beta^{[2]}$ → $\tilde{z}^{[2]}$ → $a^{[2]}$ ……

Batch norm *Batch norm*

$Parameters$: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]} \ldots\ldots, W^{[L]}, b^{[L]}$

$\gamma^{[1]}, \beta^{[1]}, \gamma^{[2]}, \beta^{[2]} \ldots\ldots, \gamma^{[L]}, \beta^{[L]}$

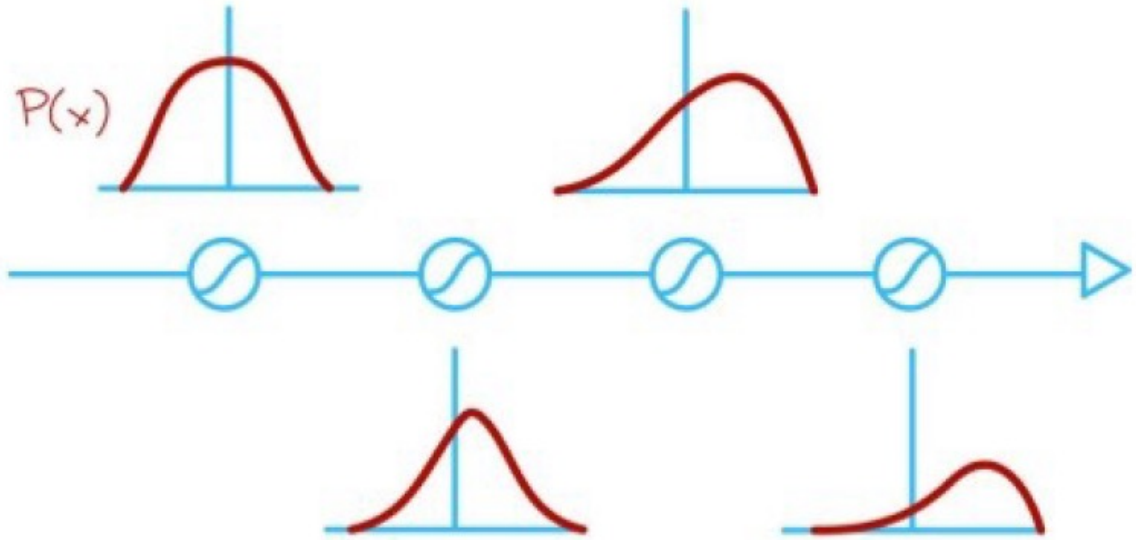
Normalizing Activations in a Network

Batch normalization

As learning progresses, the distribution of layer inputs changes due to parameter updates.

This can result in most inputs being in the nonlinear regime of the activation function and slow down learning.

Batch normalization is a technique to reduce this effect.



Ioffe and Szegedy. **Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift**, JMRL 2015

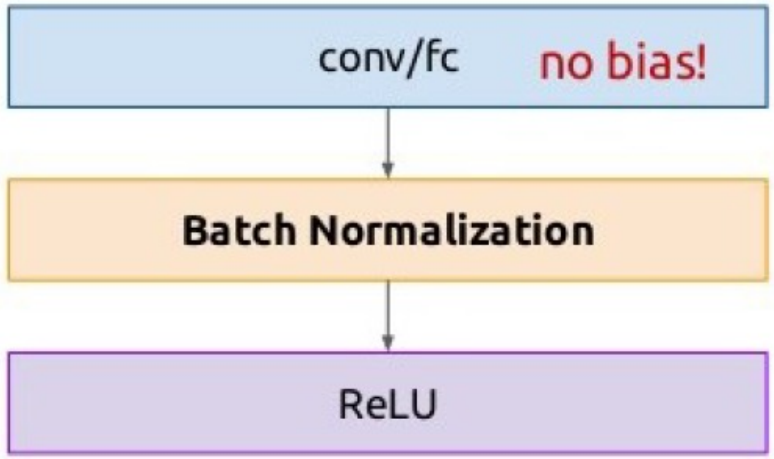
Normalizing Activations in a Network

Batch normalization

Works by **re-normalizing layer inputs to have zero mean and unit standard deviation** with respect to running batch estimates.

Also adds a **learnable scale and bias** term to allow the network to still use the nonlinearity.

Usually **allows much higher learning rates!**



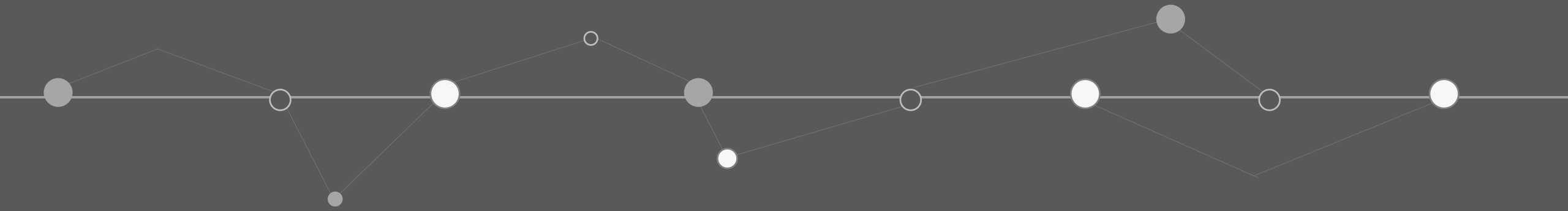
Ioffe and Szegedy. **Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift**, JMRL 2015

Why does Batch Norm Work?

Batch norm as regularization

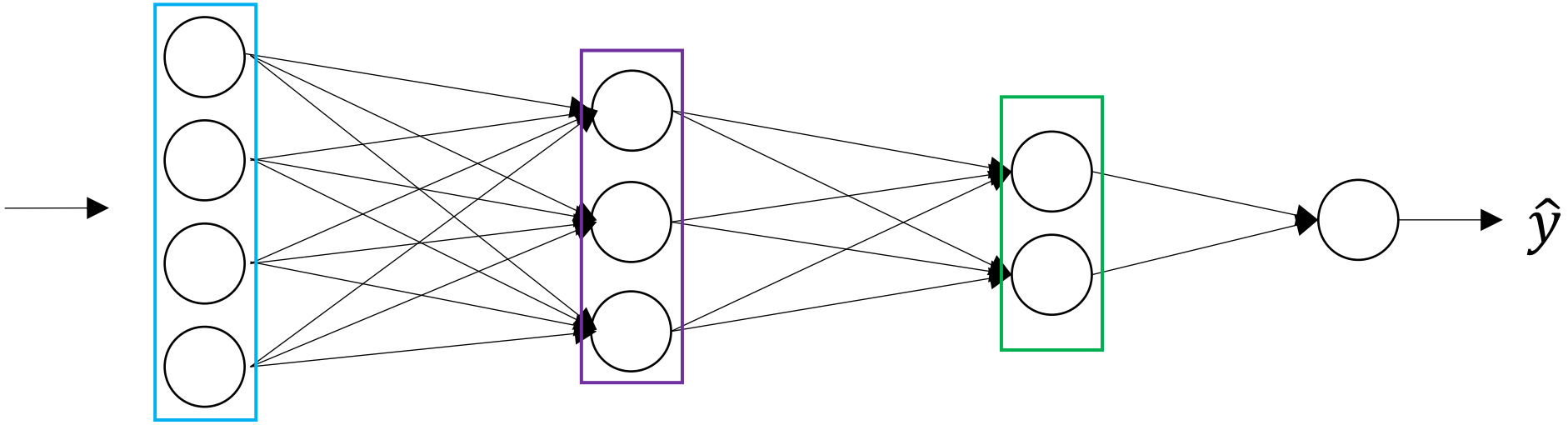
- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
- This adds some noise to the values $z^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations.
- This has a slight regularization effect.

Transfer learning

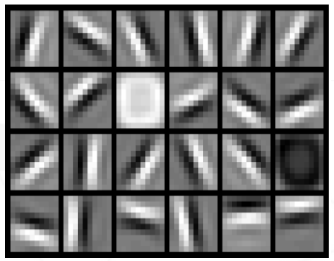


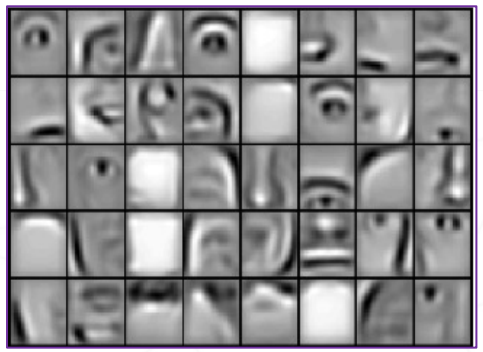
Transfer Learning

Deep learning = Learning hierarchical representations



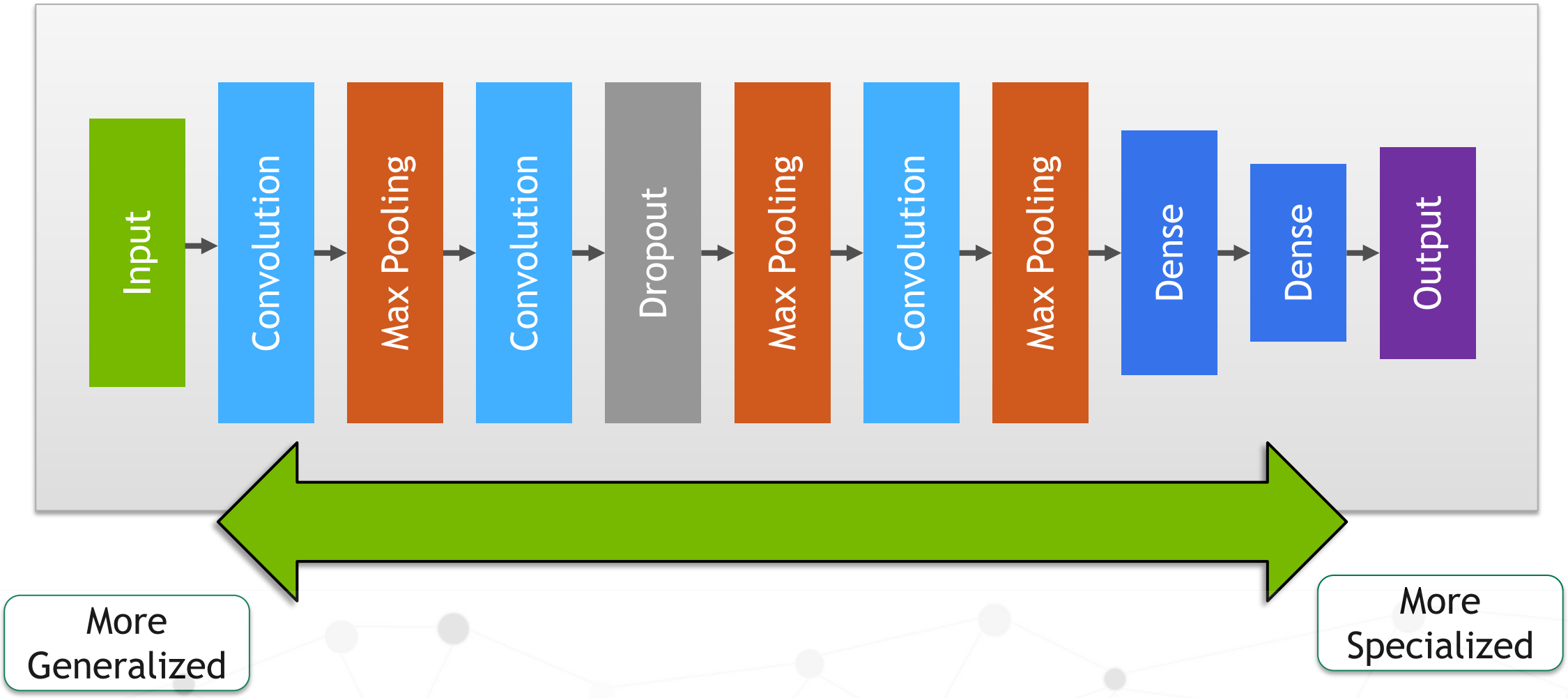
Low-level feature → Mid-level feature → High-level feature → Trainable classifier





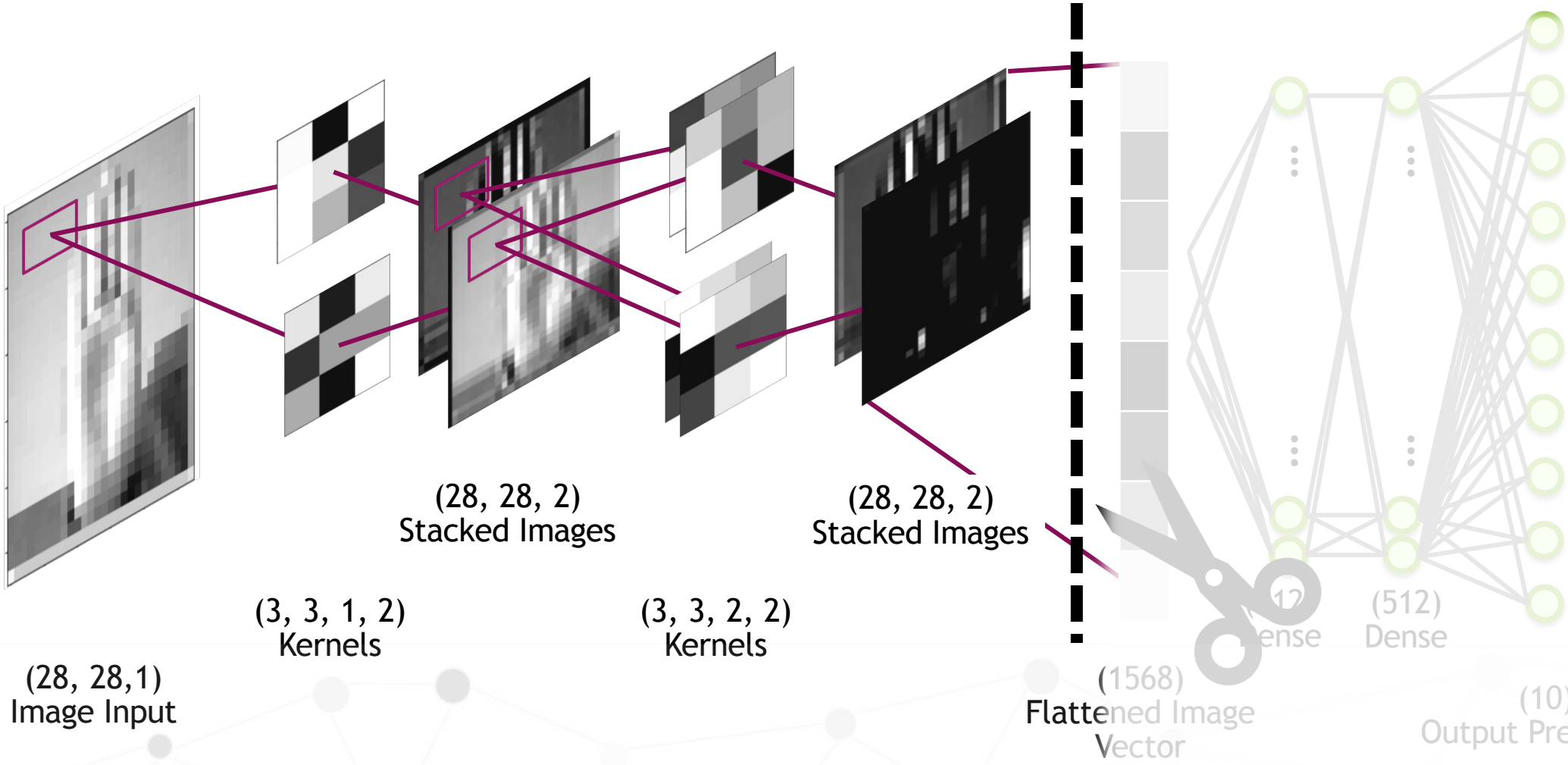
Transfer Learning

Pre-trained model



Transfer Learning

Cut the pre-trained model



Transfer Learning

Sitting on the shoulders of giants

[What Makes Transfer Learning Work for Medical Images]

[Understanding Transfer Learning for Medical Imaging]

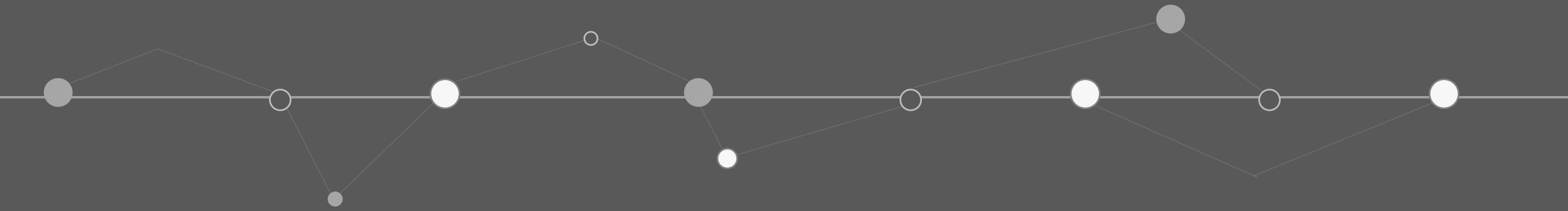
Pre-trained models

TensorFlow Hub

Keras

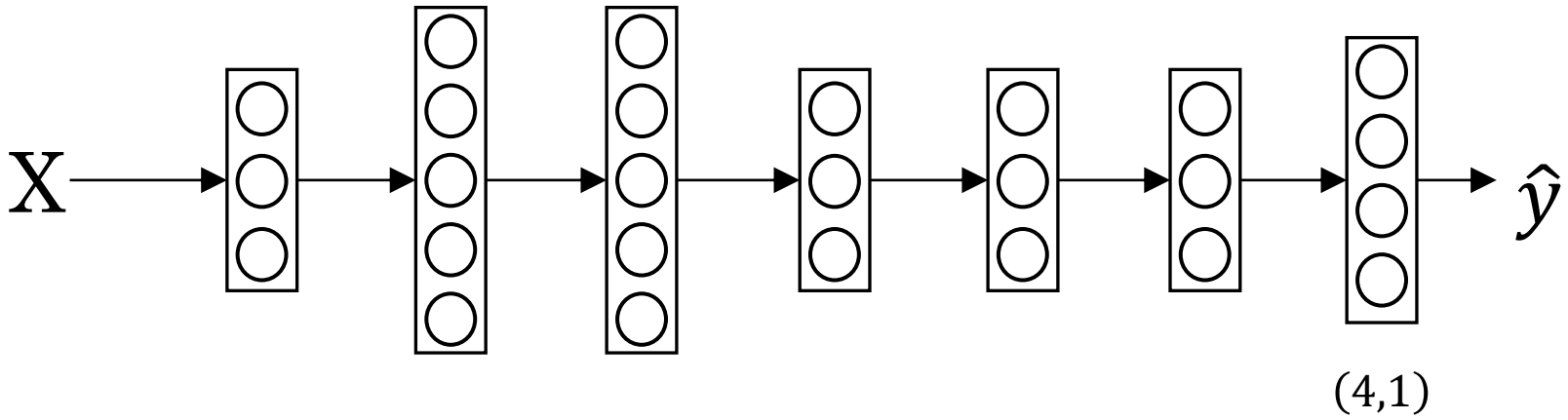
PYTORCH HUB

Multi-Class Classification



Recognizing cats, dogs, and baby chicks

3 1 2 0 3 2 0 1



(4,1)

Bird
Cat
Dog
Koala

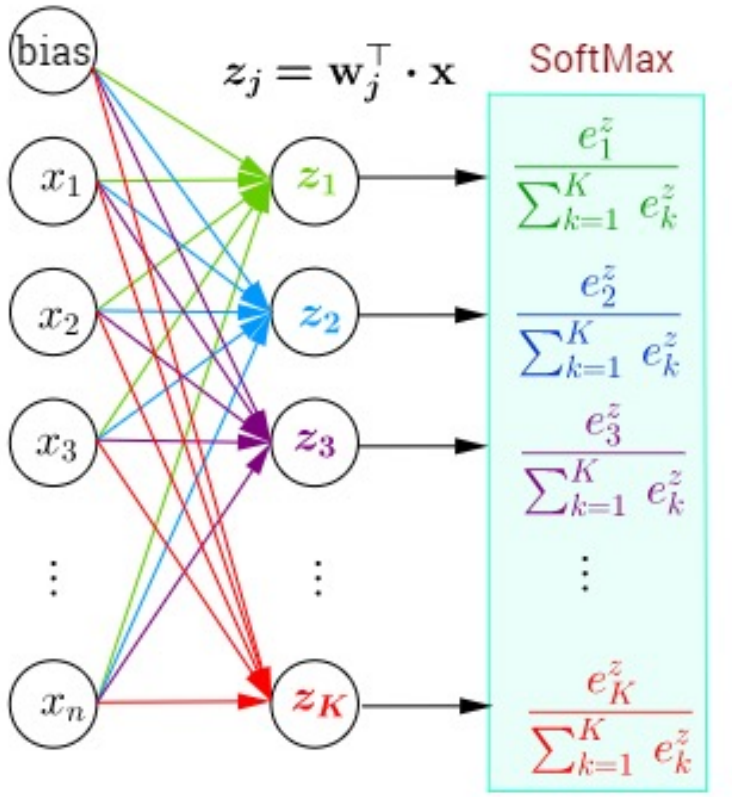
$$\begin{matrix}0\\0\\1\\0\end{matrix}$$

$y^{(i)}$

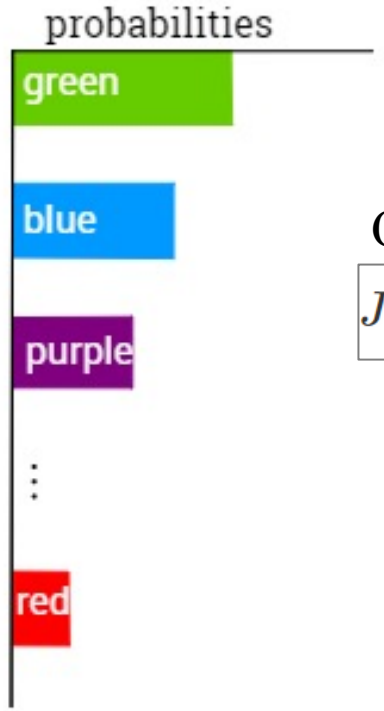
Multi-Class Classification

Softmax regression

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_K \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \\ \mathbf{w}_3^\top \\ \vdots \\ \mathbf{w}_K^\top \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$



$$S(x_j) = \frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}}, j = 1, 2, \ldots, K$$



Categorical Cross-entropy

$$J(w^{[1]}, b^{[1]} \ldots) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)})$$

$$L(\hat{y}^{(i)}, y^{(i)}) = -\sum_{i=1}^{k} y_i \cdot \log \hat{y}_i$$

Binary classification (k=2):

$$p(y=1|x) = \frac{e^{\theta_1^T x}}{e^{\theta_0^T x} + e^{\theta_1^T x}} = \frac{1}{1 + e^{(\theta_0^T - \theta_1^T)x}} = \frac{1}{1 + e^{-\beta x}}$$

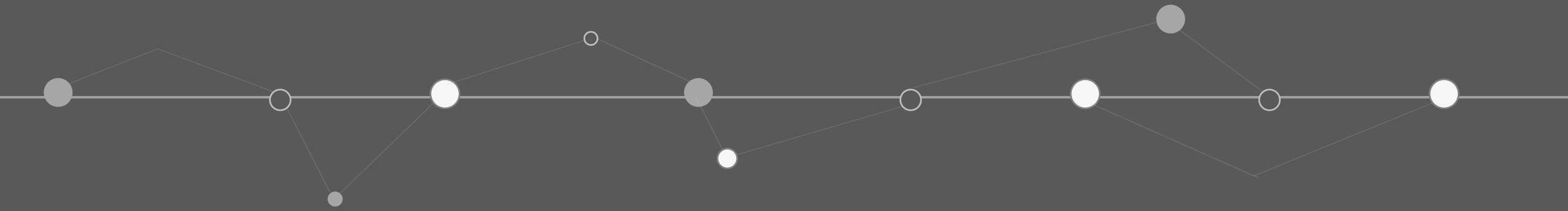
$$p(y=0|x) = \frac{e^{\theta_0^T x}}{e^{\theta_0^T x} + e^{\theta_1^T x}} = \frac{e^{(\theta_0^T - \theta_1^T)x}}{1 + e^{(\theta_0^T - \theta_1^T)x}} = \frac{e^{-\beta x}}{1 + e^{-\beta x}}$$

$$\beta = -(\theta_0^T - \theta_1^T)$$

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$J(w,b) = \frac{1}{m}\sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^m [(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$$

Multi-Task Learning



Multi-Task Learning

Simplified autonomous driving example

$X^{(i)}$

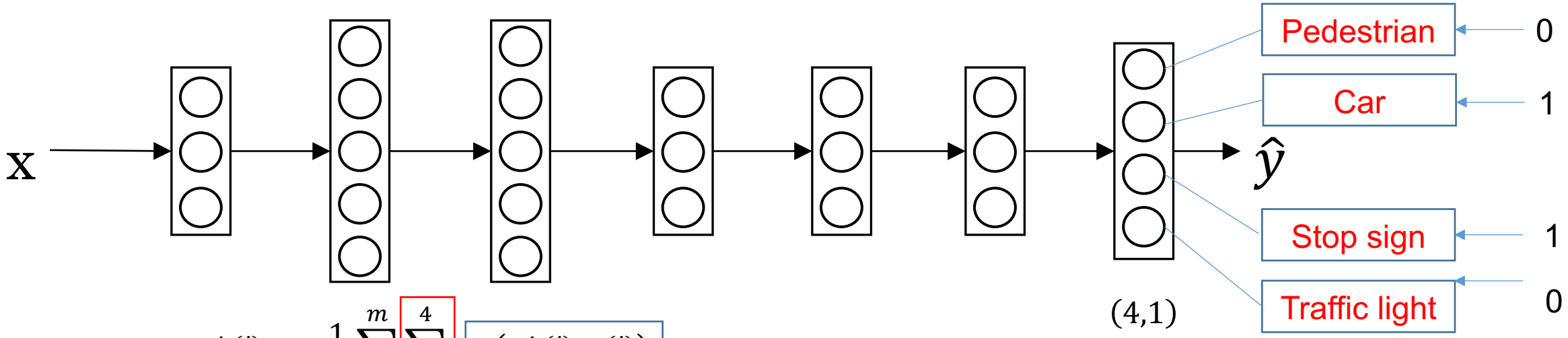
$y^{(i)}$ $(4, m)$

Pedestrian	0
Car	1
Stop sign	1
Traffic light	0
.	.
.	.
.	.

$$Y = \begin{bmatrix} | & | & | & & | \\ y^{(1)} & y^{(2)} & y^{(3)}, & \ldots, & y^{(m)} \\ | & | & | & & | \end{bmatrix}$$

Multi-Task Learning

Neural network architecture



$$Loss: \hat{y}^{(i)}_{(4,1)} \rightarrow \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4} L\left(\hat{y}_j^{(i)}, y_j^{(i)}\right)$$

Usually logistic loss

$$-y_j^{(i)} \log \hat{y}_j^{(i)} - \left(1 - y_j^{(i)}\right) \log\left(1 - \hat{y}_j^{(i)}\right)$$

Sum only over value of j with 0/1 label

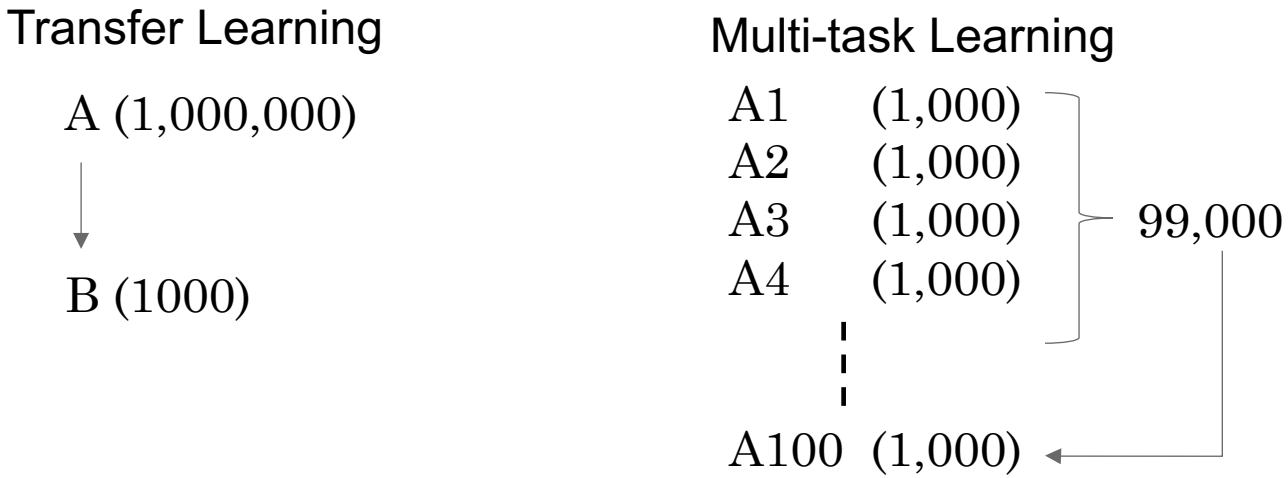
Unlike Softmax regression:
One image can have multiple labels

Multi-task Learning

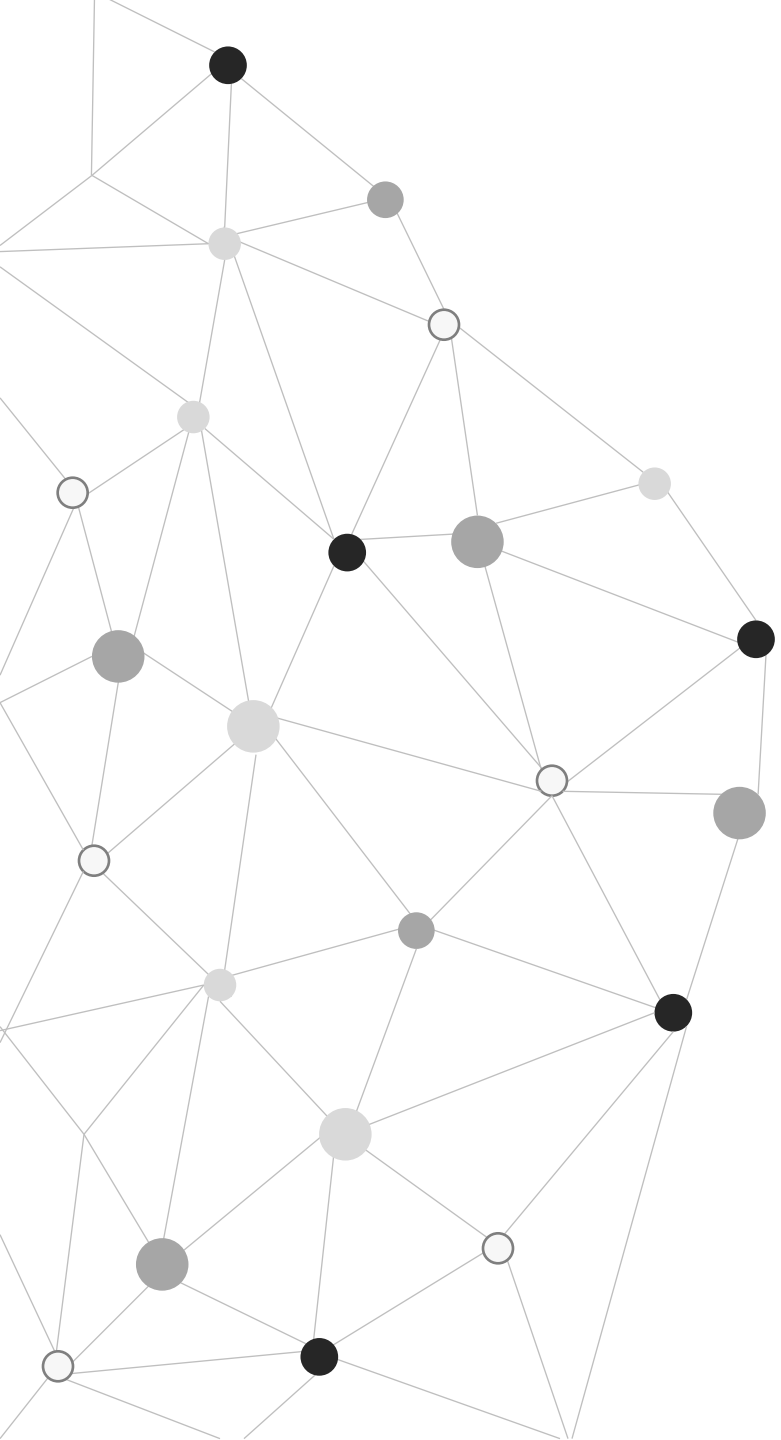
Multi-Task Learning

When multi-task learning makes sense

- Training on a set of tasks that could benefit from having shared lower-level features.
- Usually: Amount of data you have for each task is quite similar.



- Can train a big enough neural network to do well on all the tasks.



Next:

Lab Practice

Build a CNN