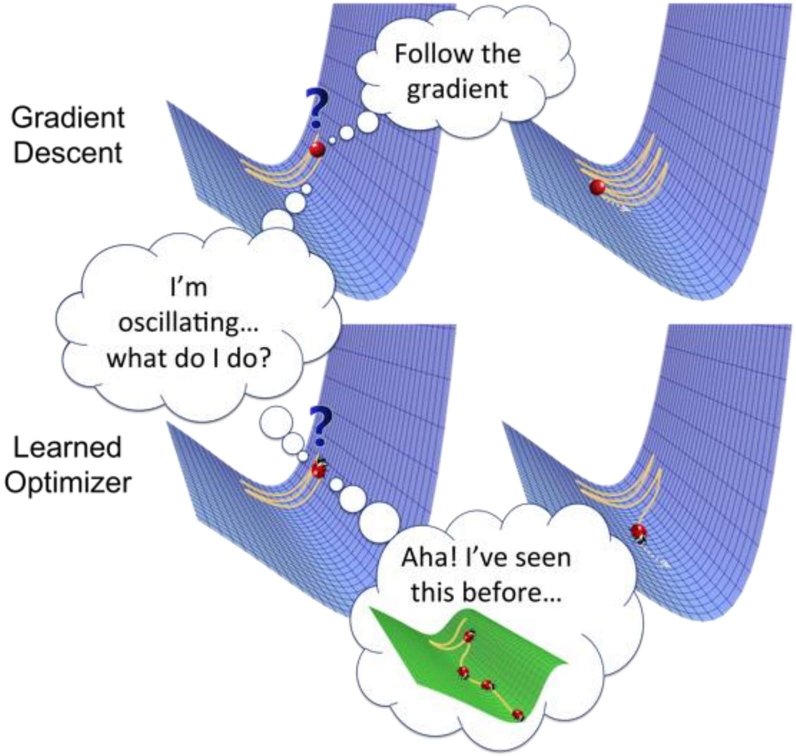
NEURAL NETWORK

Why Neural Networks now?

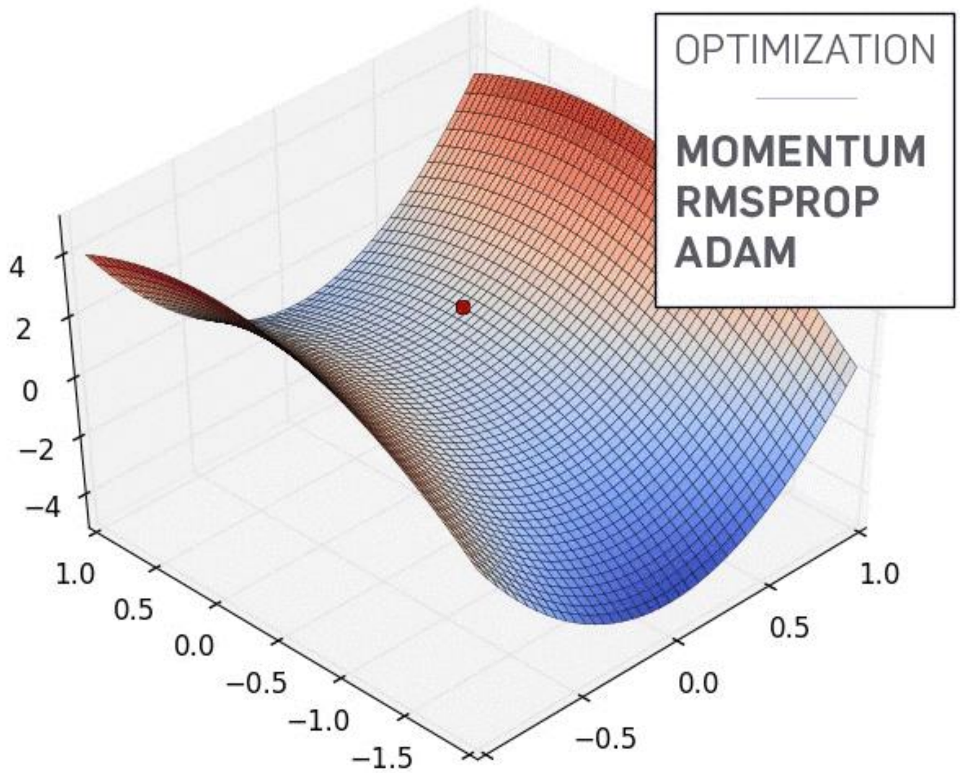
- Better activation functions for neural layers
- Better weight-initialization schemes
- Better optimization schemes
- Faster computations

Optimizers and Parameters

- Optimization ~ change the weights and learning rate in order to reduce the loss function.
- More than one way ~ Why do we need more?



Optimizers and Parameters

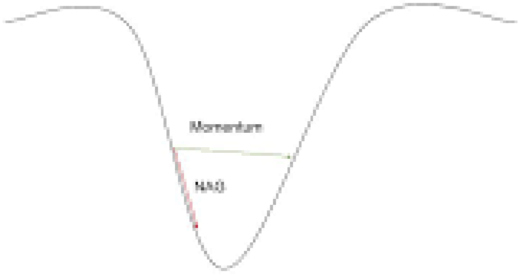


Optimizers and Parameters

- Gradient Decent ~ The Most Basic
- Stochastic Gradient Descent
- Mini Batch SGD
- Momentum
  - GD: $\theta=\theta-\alpha\cdot\nabla J(\theta)$
  - Mom: $V(t)=\gamma V(t-1)+\alpha.\nabla J(\theta)$, $\theta=\theta-V(t)$.

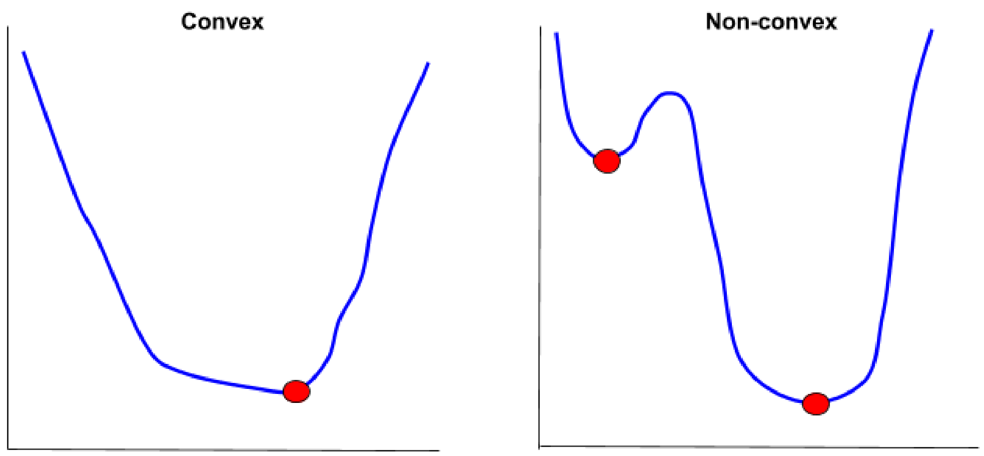
Optimizers and Parameters

- Nesterov Accelerated Gradient



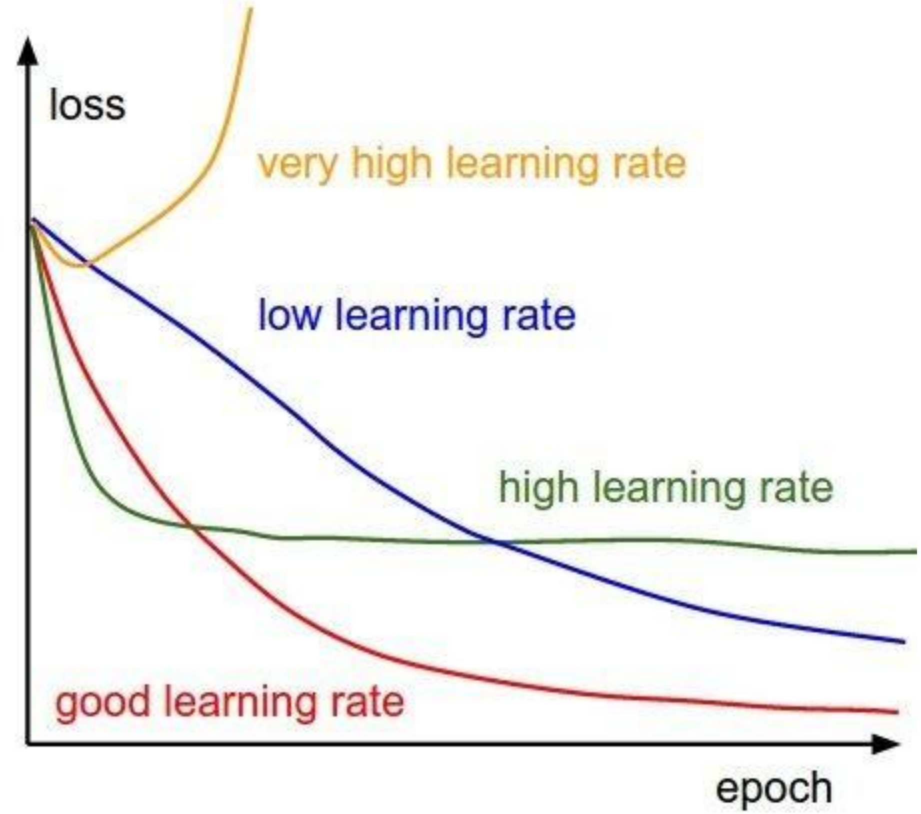
Learning Rate

- Gradient Decent ~ Learning Rate
- Optimal value for the learning rate, eta (η).
  - too small ~ too slow
  - too large ~ overshoot, no convergence
- No “optimal” solution



Learning rate selection

The effects of step size (or “learning rate”)



Learning Rate

- Fixed Learning Rate (usually small, say 0.1 or 0.01)
- Annealing Learning Rate
- Cyclical Learning Rate
- Adaptive Learning Rate

Learning Rate

- How to determine?
- Start with a traditional default value
- Use Diagnostic plots
- Sensitivity Analysis (Hyperparameter tuning)
- Adaptive Learning Rate

Momentum

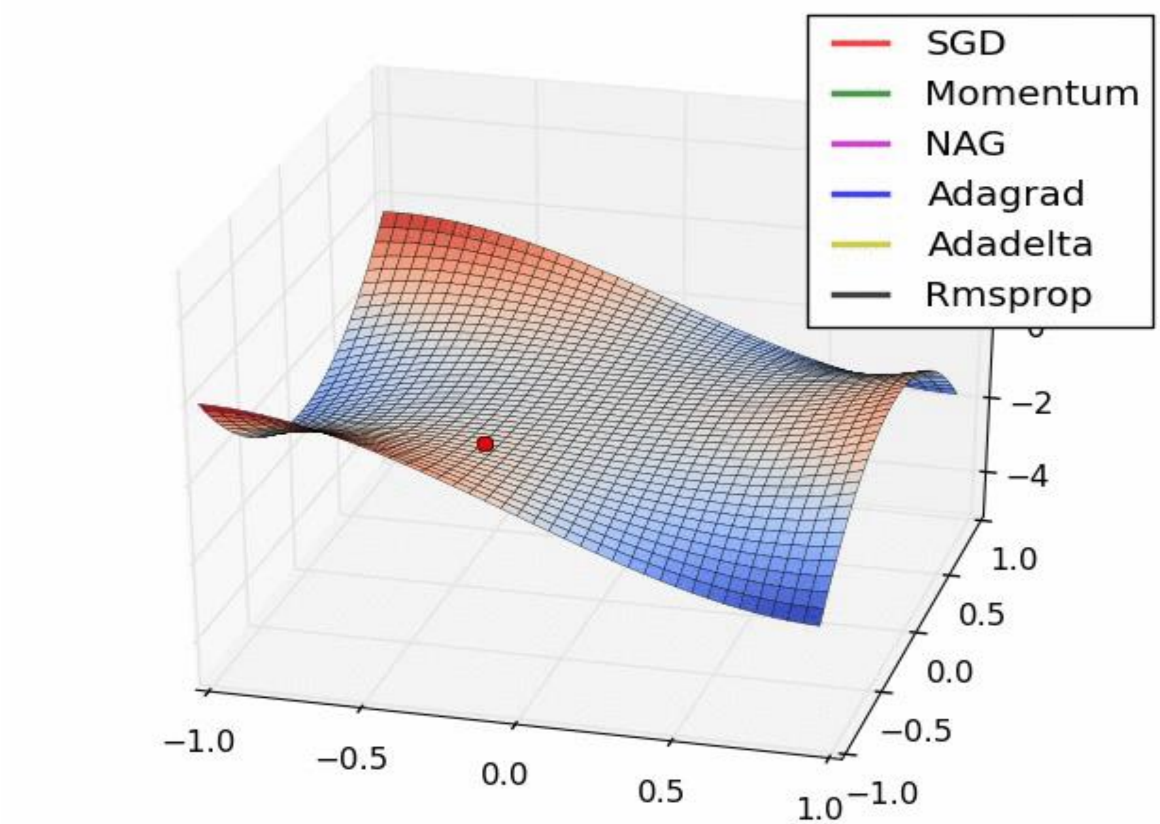
- Including an exponentially weighted average of the prior updates while updating weights
- Motivation is to cause updating in the same direction in the future.
- Momentum is set to a value greater than 0.0 and less than one, where common values such as 0.5, 0.9 and 0.99 are used in practice.

Optimizers and Parameters

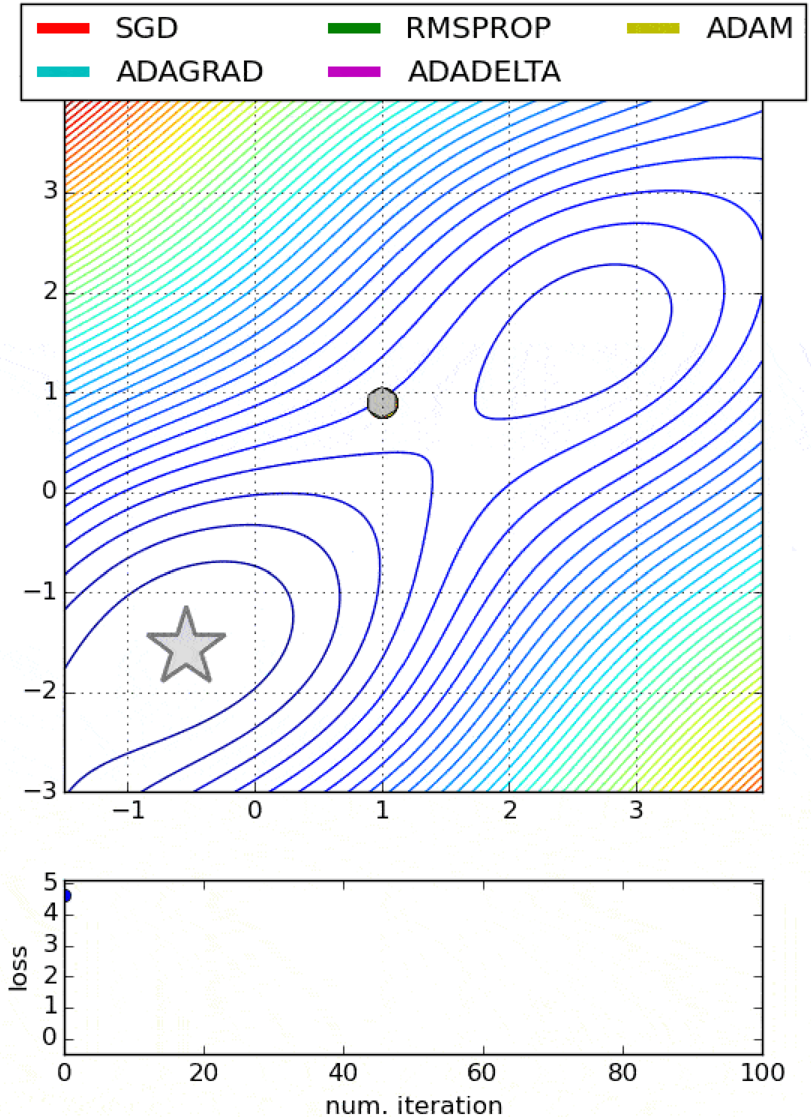
- Adagrad
  - Non-constant learning rate
- AdaDelta
  - Improvement of Adagad
  - Nondecaying learning rate

Optimizers and Parameters

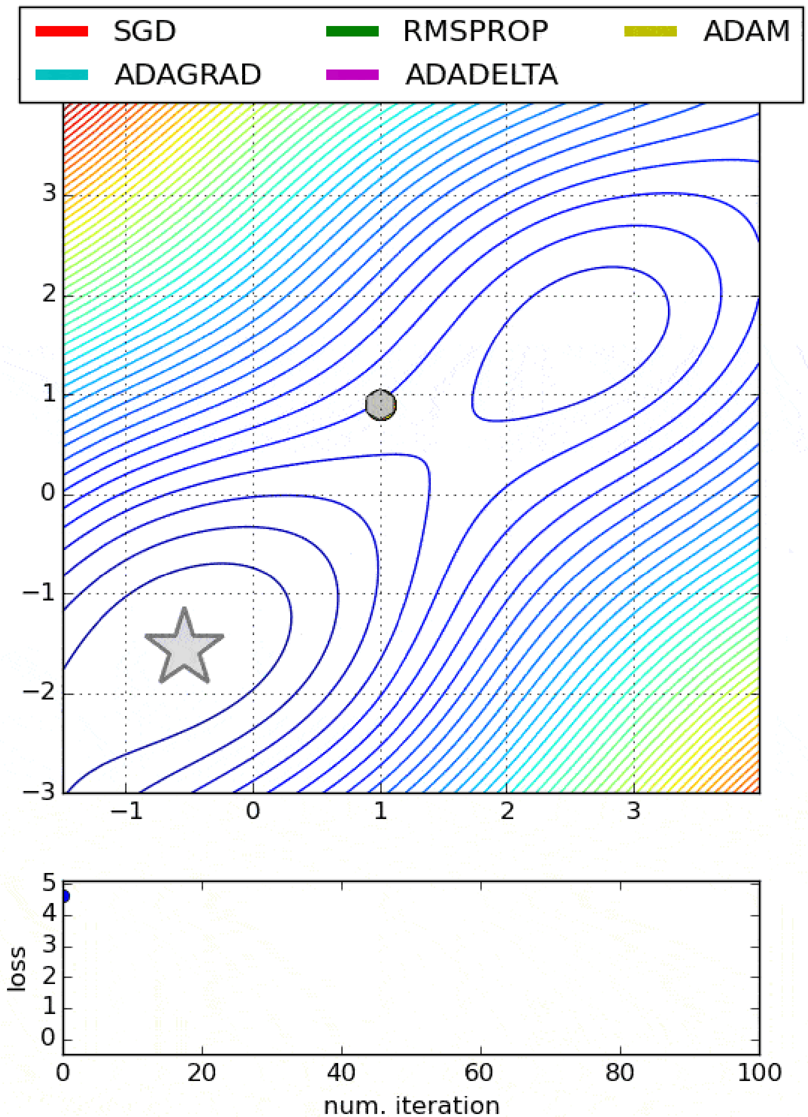
- RMSProp
- Adam
  - Adaptive Moment Estimation



Optimizers and Parameters



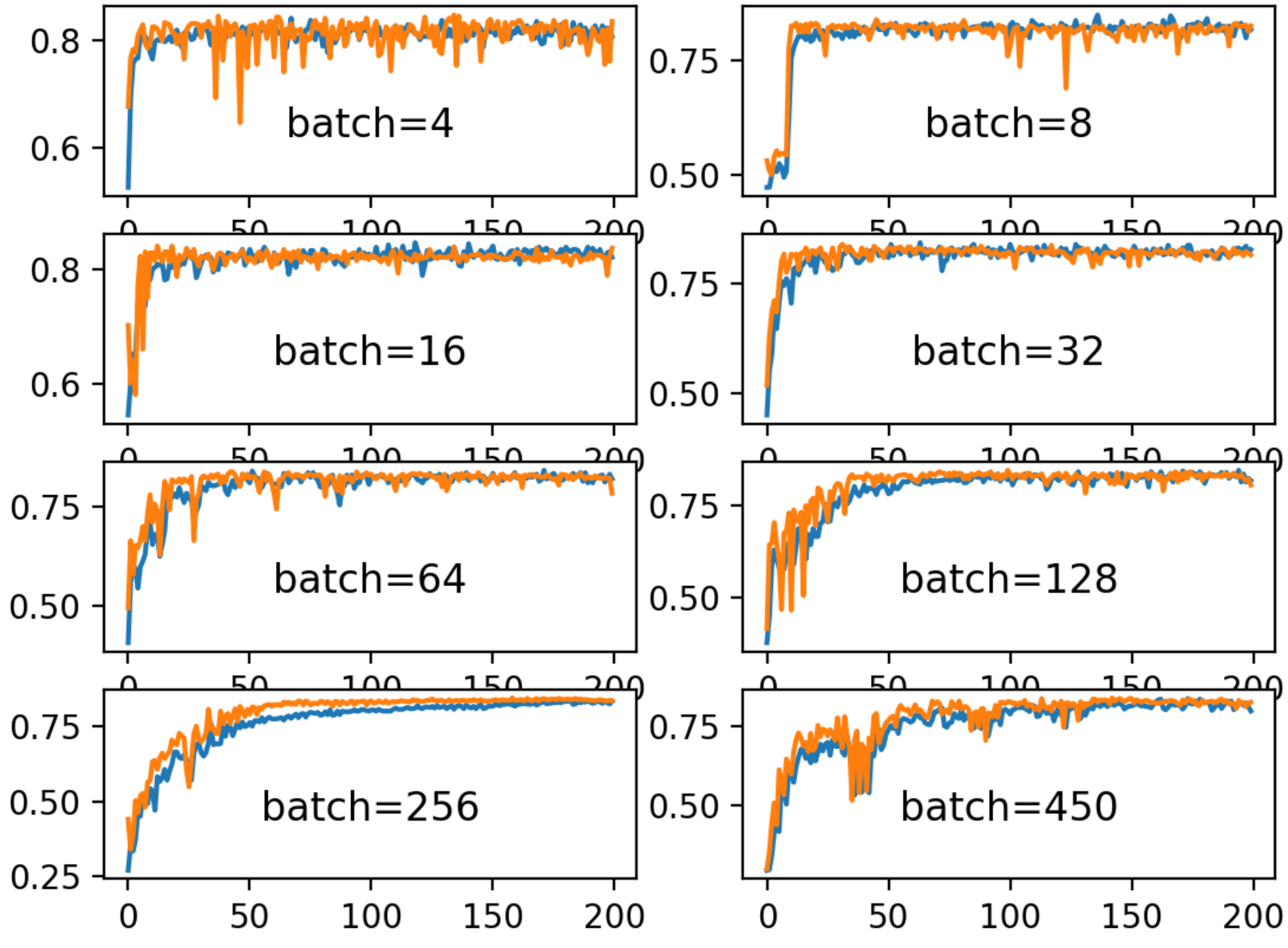
Optimizers and Parameters



Batch Size Selection

- Batch size controls the accuracy of the estimate of the error gradient
- Batch, Stochastic, and Minibatch gradient descent
- Tradeoff in determining the batch size ~ the speed and stability of the learning process.

Batch Size Selection

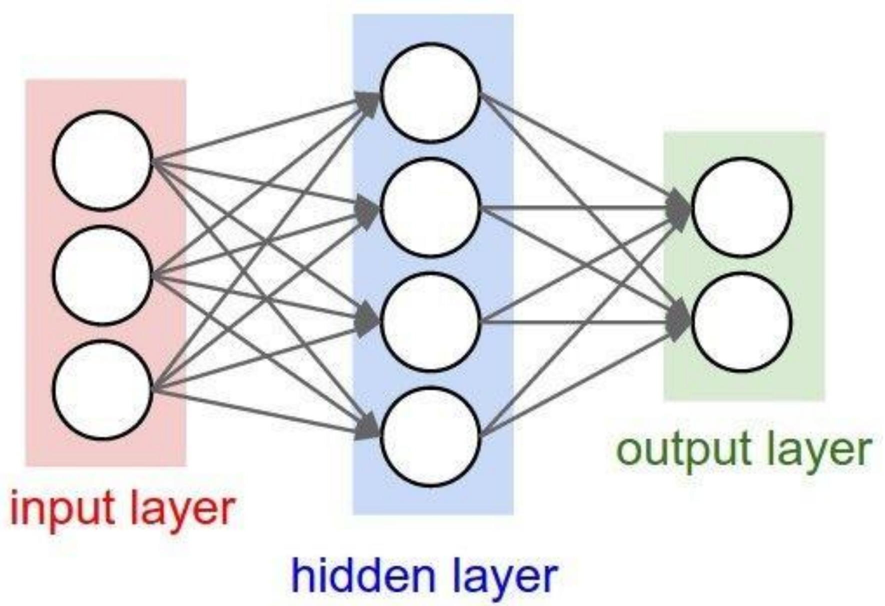


Network Structure

- Number of layers and size of each layer in NN is important
- Used to have shallow networks but recently deep networks are popular
- Too large networks (both size and number of layers) tend to overfit, why?

Weight Initialization

- Weight initialization ~ updating the parameters normally
- Vanishing or exploding gradients



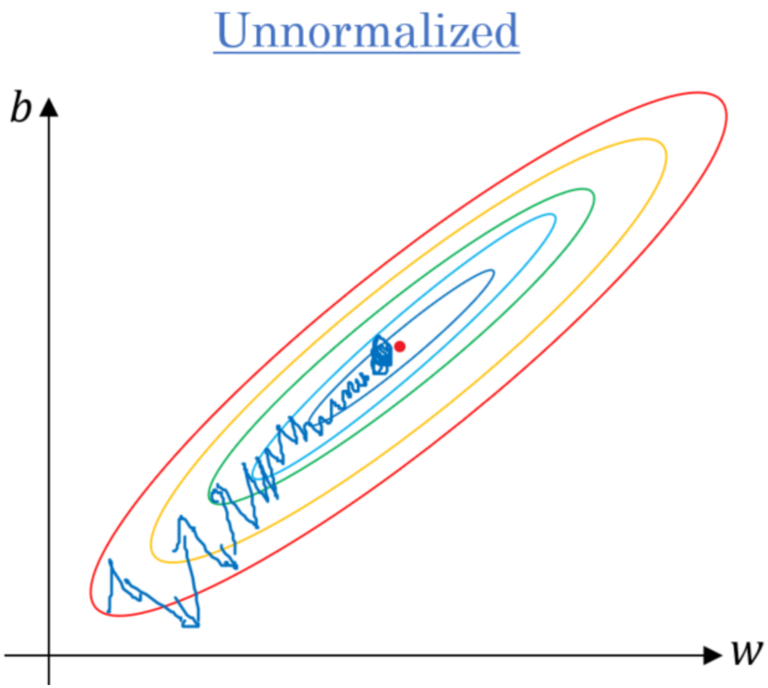
Weight Initialization

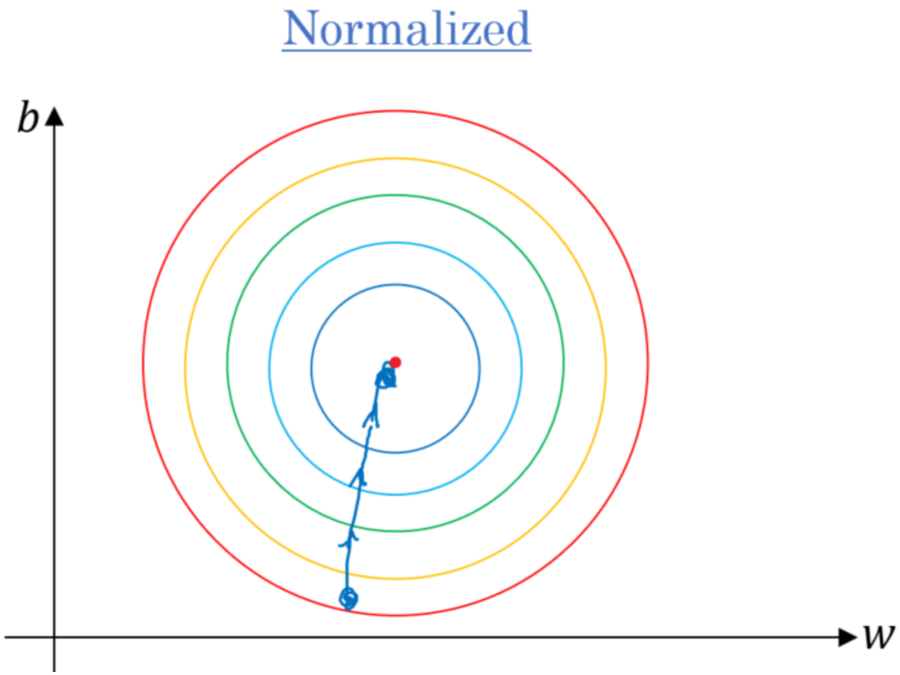
- Zero Initialization
- Random Initialization
- Xavier Initialization
- He Initialization

Preprocessing the Data

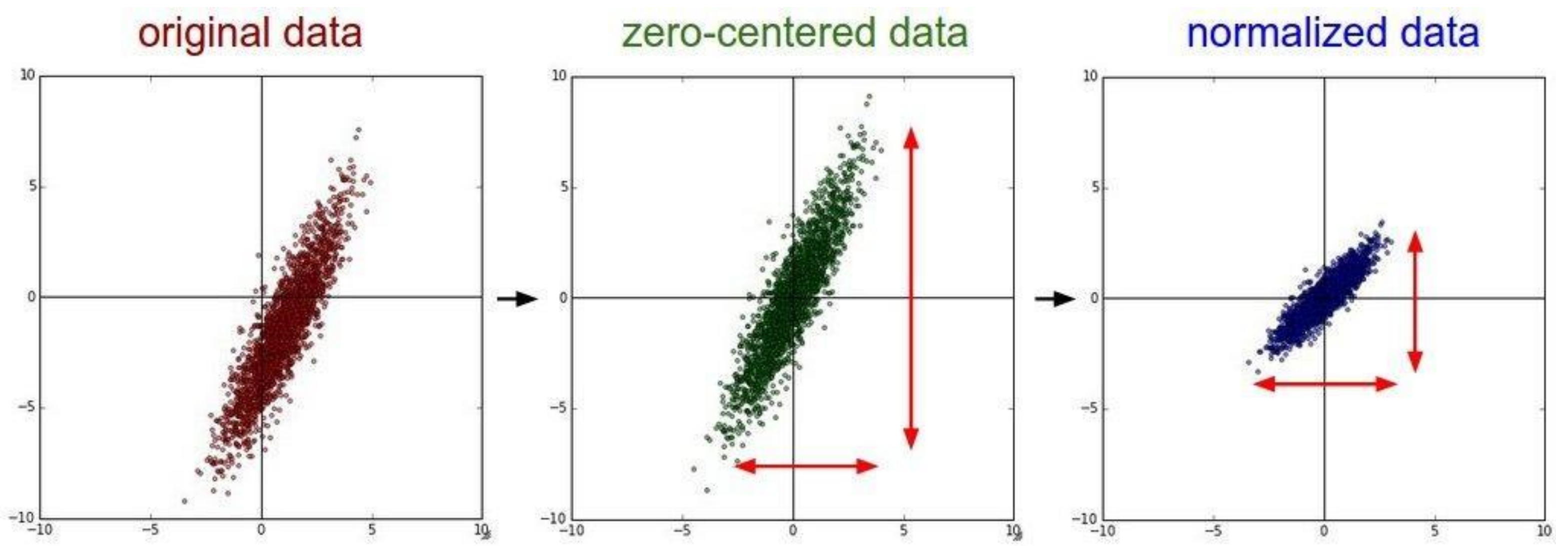
- Scaling the input data is important
- Large Values ~ Large Weights ~ Unstable model
- Rule of Thumb ~ range 0-1 or standardized with zero mean and stan. dev. of 1.

Scaling the Data





Preprocessing the Data




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

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

(Assume X [NxD] is data matrix,
each example in a row)

Preprocessing the Data

- Data Normalization

Preprocessing the Data

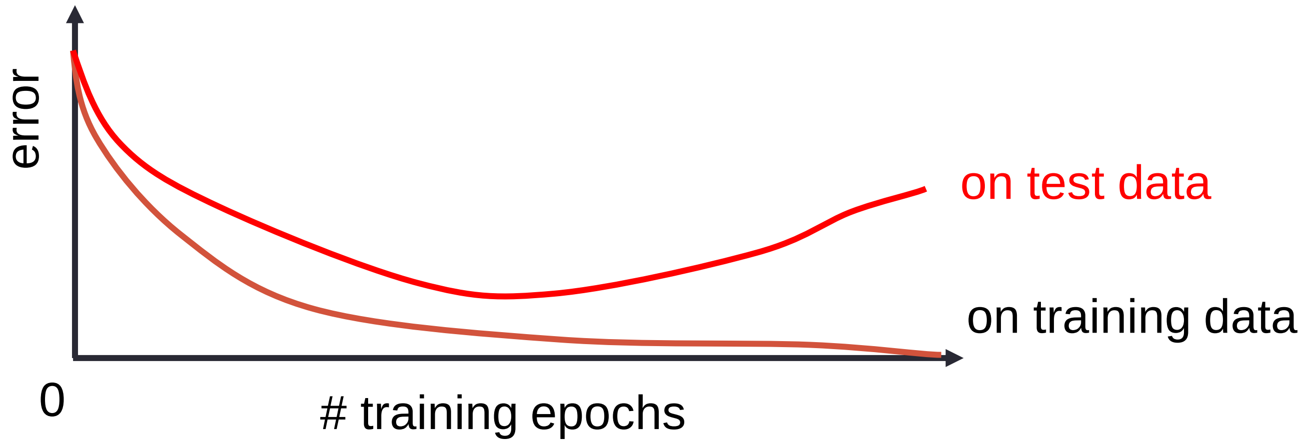
- Data Standardization

Overfitting

- Deep neural networks require lots of data, and can overfit easily
- The more weights you need to learn, the more data you need
- That’s why with a deeper network, you need more data for training than for a shallower network
- Ways to prevent overfitting include:
  - Using a validation set to stop training or pick parameters
  - Regularization
  - Data Augmentation

Over-training prevention

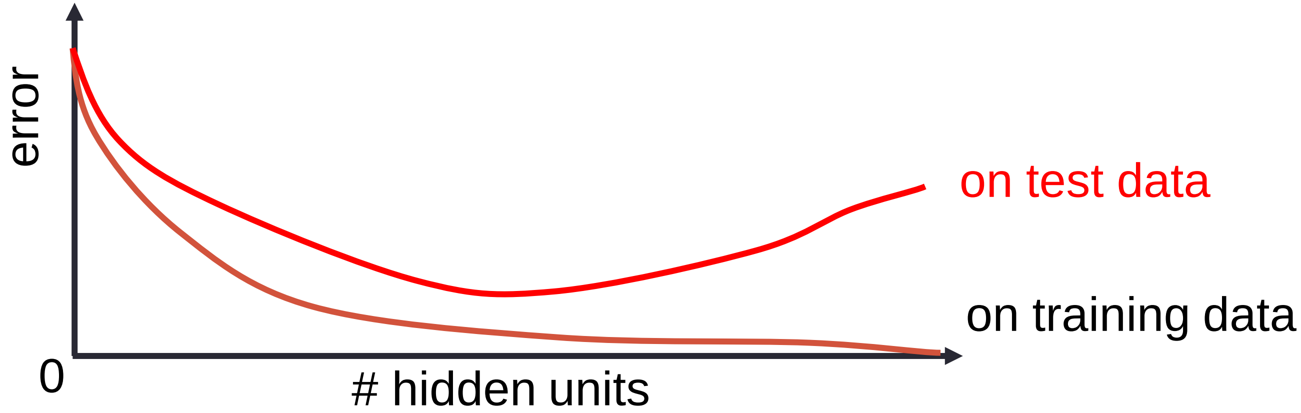
- Running too many epochs can result in over-fitting.



- Keep a hold-out validation set and test accuracy on it after every epoch. Stop training when additional epochs actually increase validation error.

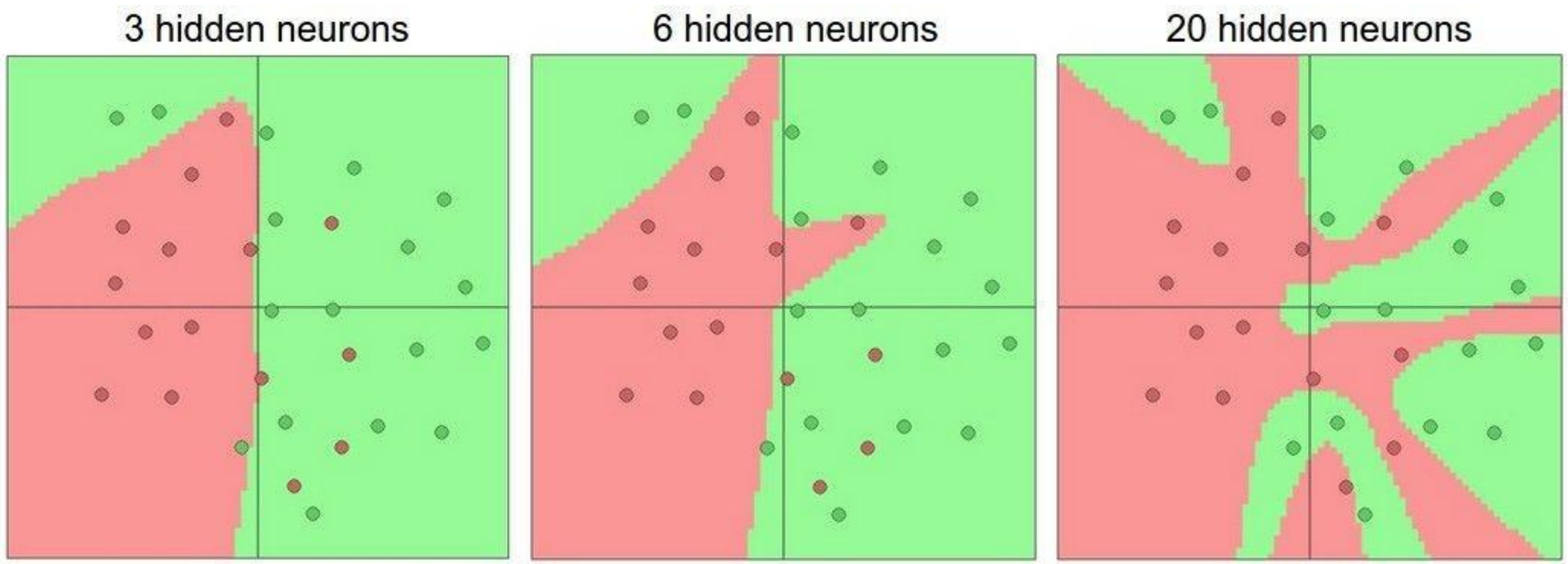
Determining best number of hidden units

- Too few hidden units prevents the network from adequately fitting the data.
- Too many hidden units can result in over-fitting.



- Use internal cross-validation to empirically determine an optimal number of hidden units.

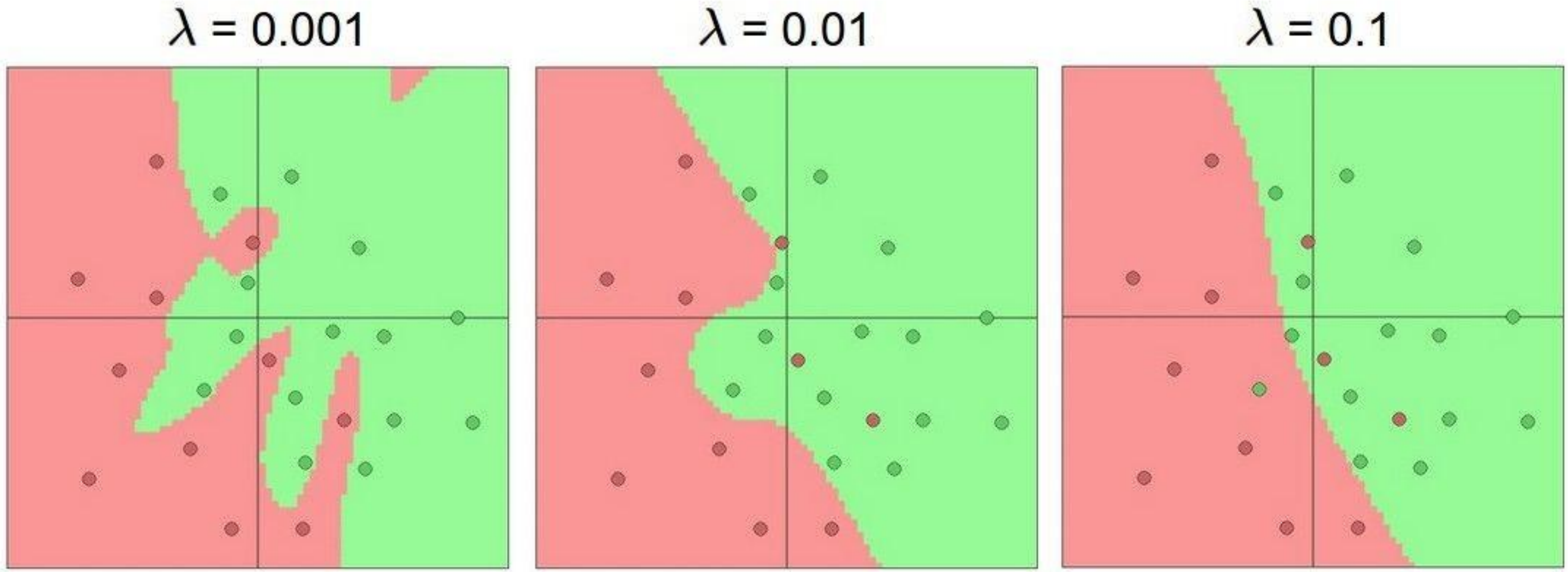
Effect of number of neurons



more neurons = more capacity

Effect of regularization

Do not use size of neural network as a regularizer. Use stronger regularization instead:



(you can play with this demo over at ConvNetJS: http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html)

Weight Regularization

λ = regularization strength (hyperparameter)

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Simple examples

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

More complex:

Dropout

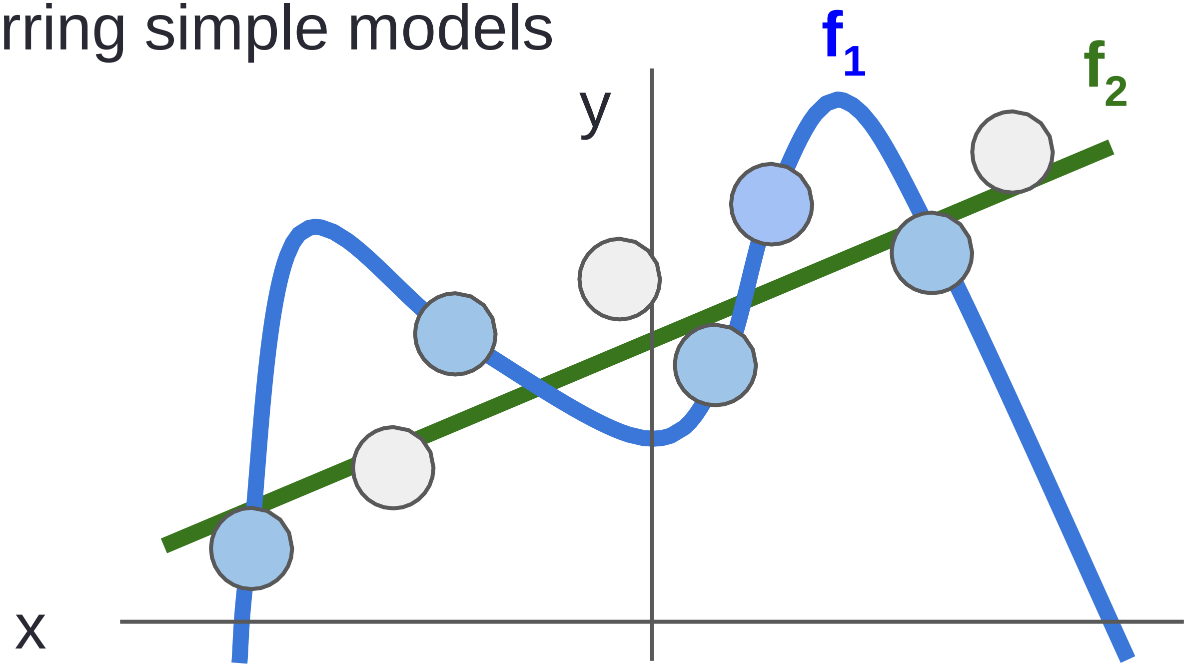
Batch normalization

Why regularize?

- Express preferences over weights
- Make the model *simple* so it works on test data

Weight Regularization

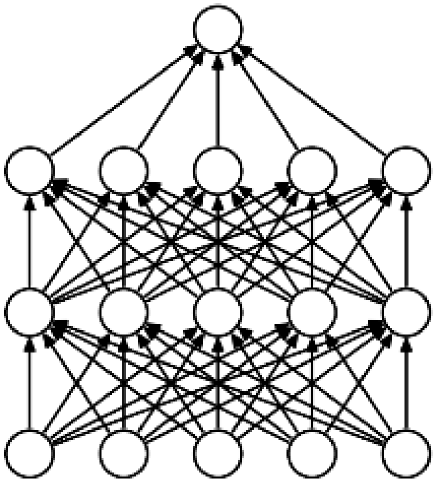
- Preferring simple models



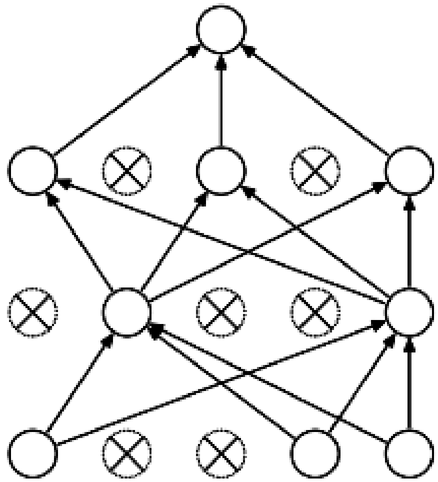
Regularization pushes against fitting the data *too* well so we don't fit noise in the data

Dropout

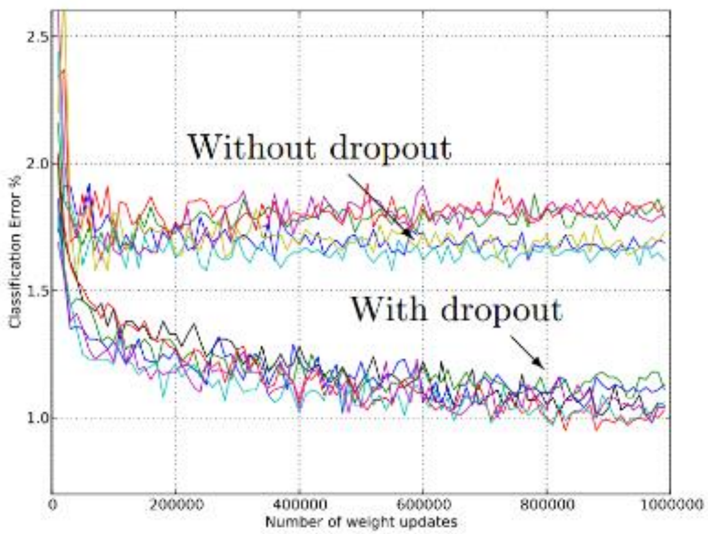
- Dropout
  - Randomly turn off some neurons
  - Allows individual neurons to independently be responsible for performance



(a) Standard Neural Net



(b) After applying dropout.



Dropout: A simple way to prevent neural networks from overfitting

Batch Normalization

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

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

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

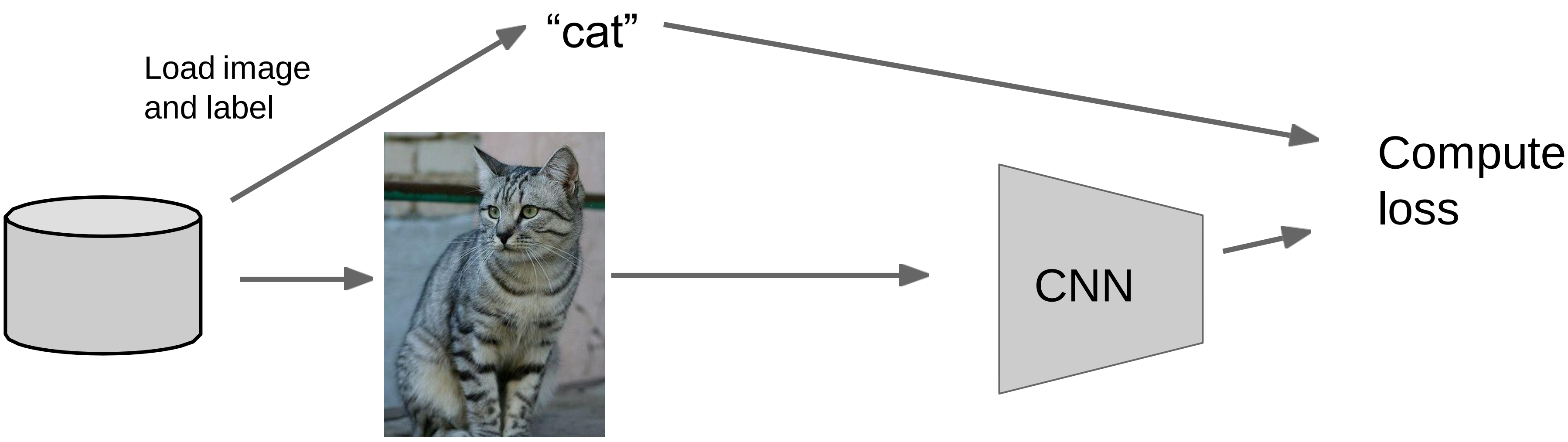
$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

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

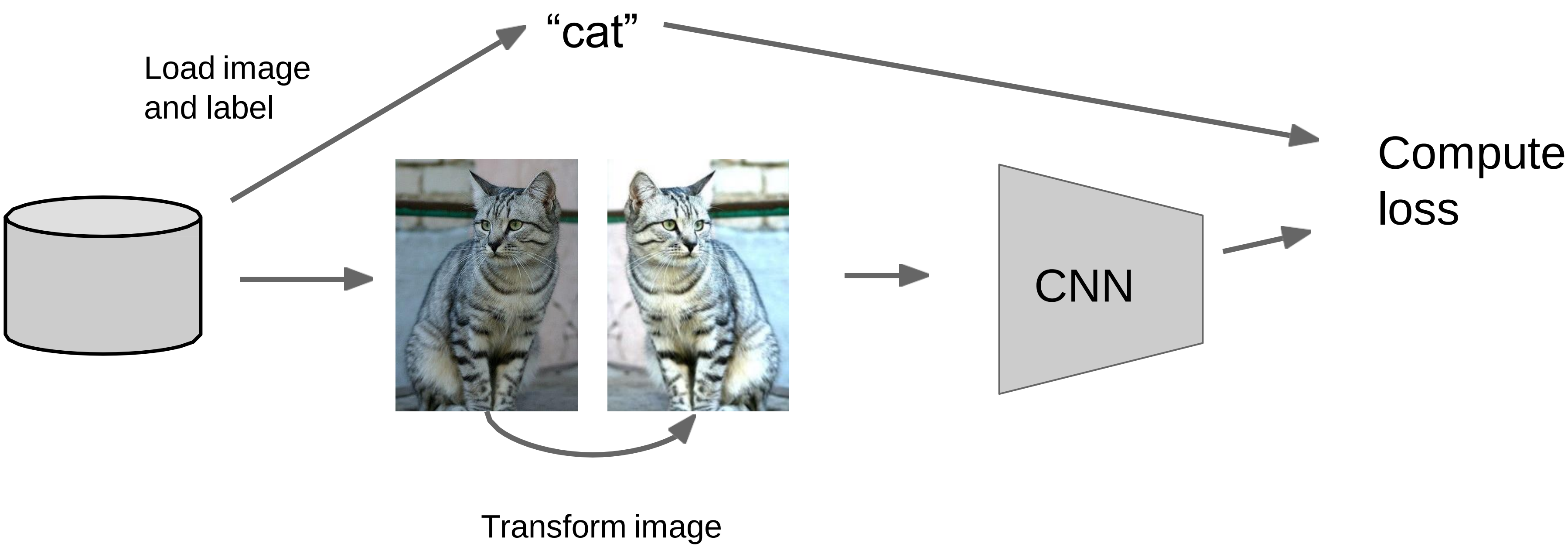
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization

Data Augmentation

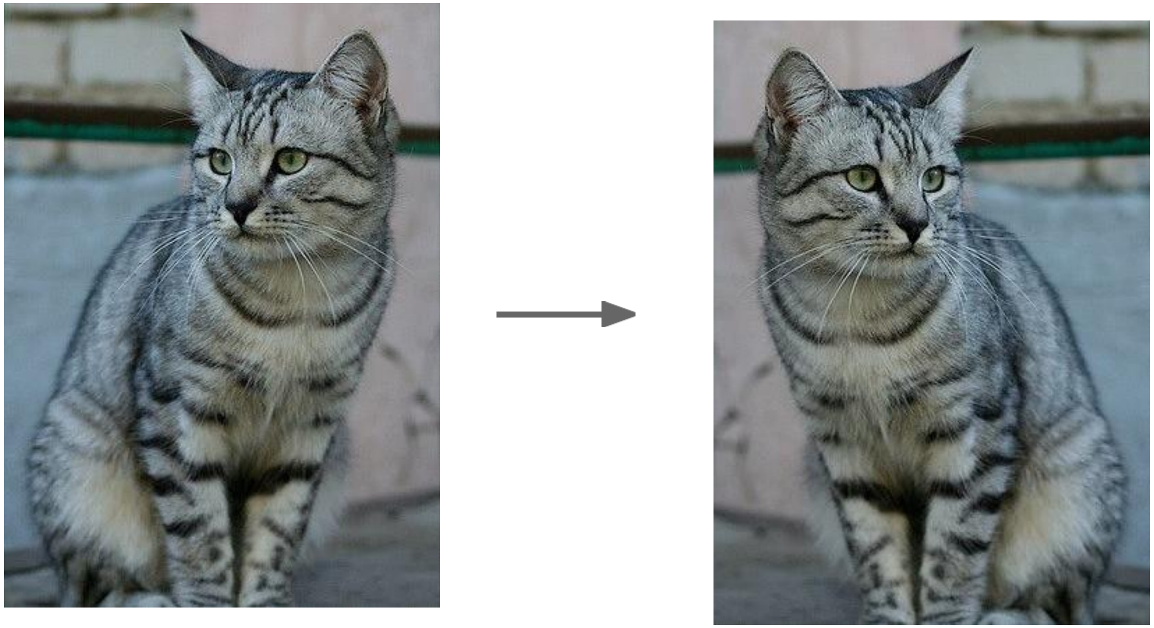


Data Augmentation



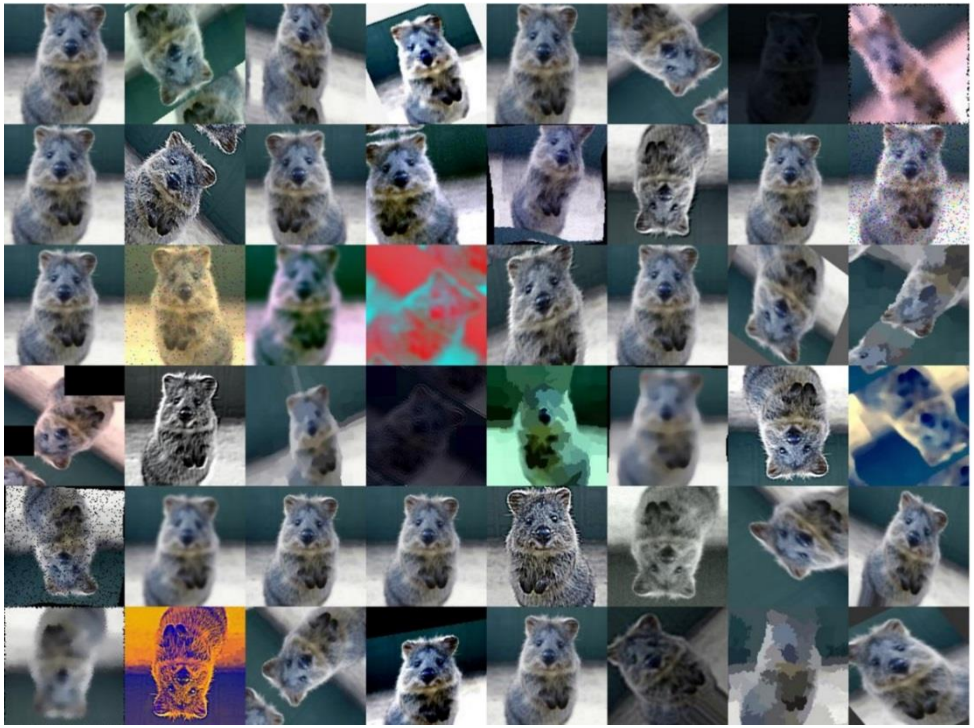
Data Augmentation

- Horizontal Flips

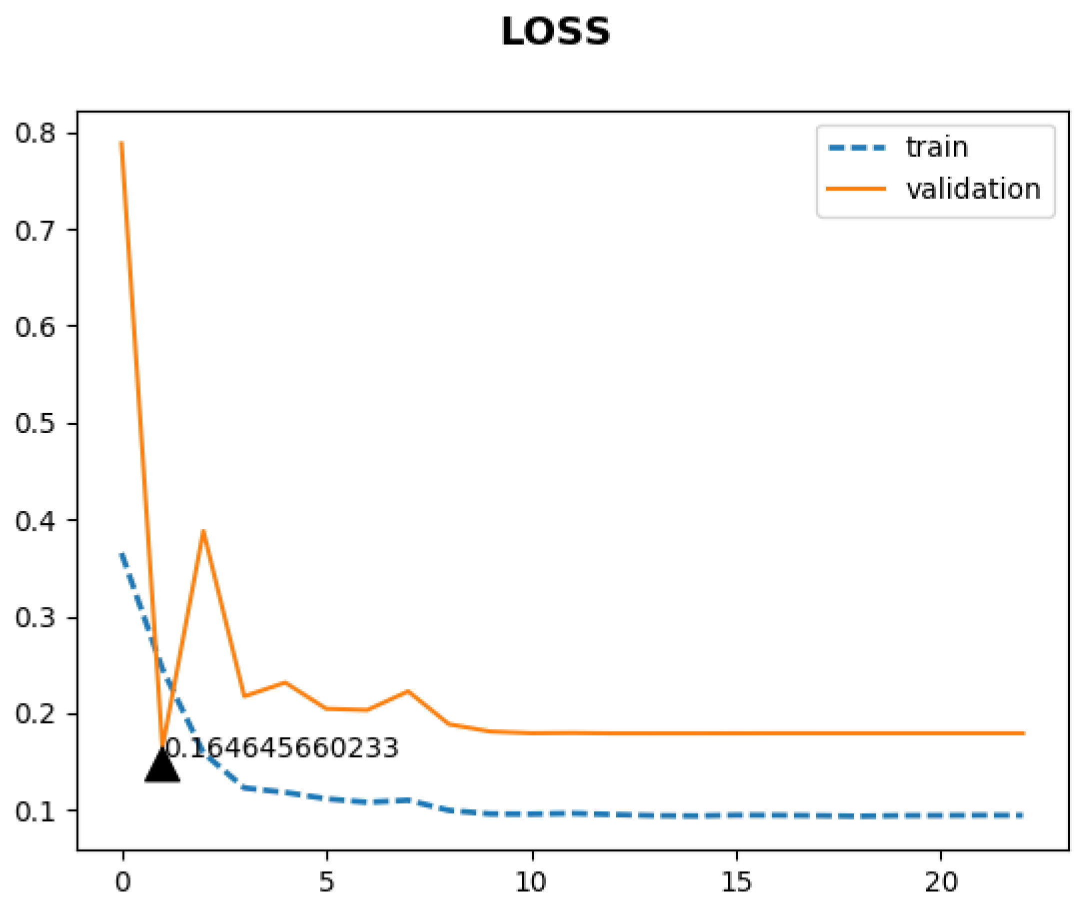


Data Augmentation

- Get creative for your problem!
- Random mix/combinations of :
  - translation
  - rotation
  - stretching
  - shearing,
  - lens distortions
  - …



Callbacks


LOSS
train
validation
0.164645660233
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
5
10
15
20

CPU vs GPU

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	$385	~540 GFLOPs FP32
GPU (NVIDIA RTX 2080 Ti)	3584	1.6 GHz	11 GB GDDR6	$1199	~13.4 TFLOPs FP32
TPU NVIDIA TITAN V	5120 CUDA, 640 Tensor	1.5 GHz	12GB HBM2	$2999	~14 TFLOPs FP32 ~112 TFLOP FP16
TPU Google Cloud TPU	?	?	64 GB HBM	$4.50 per hour	~180 TFLOP

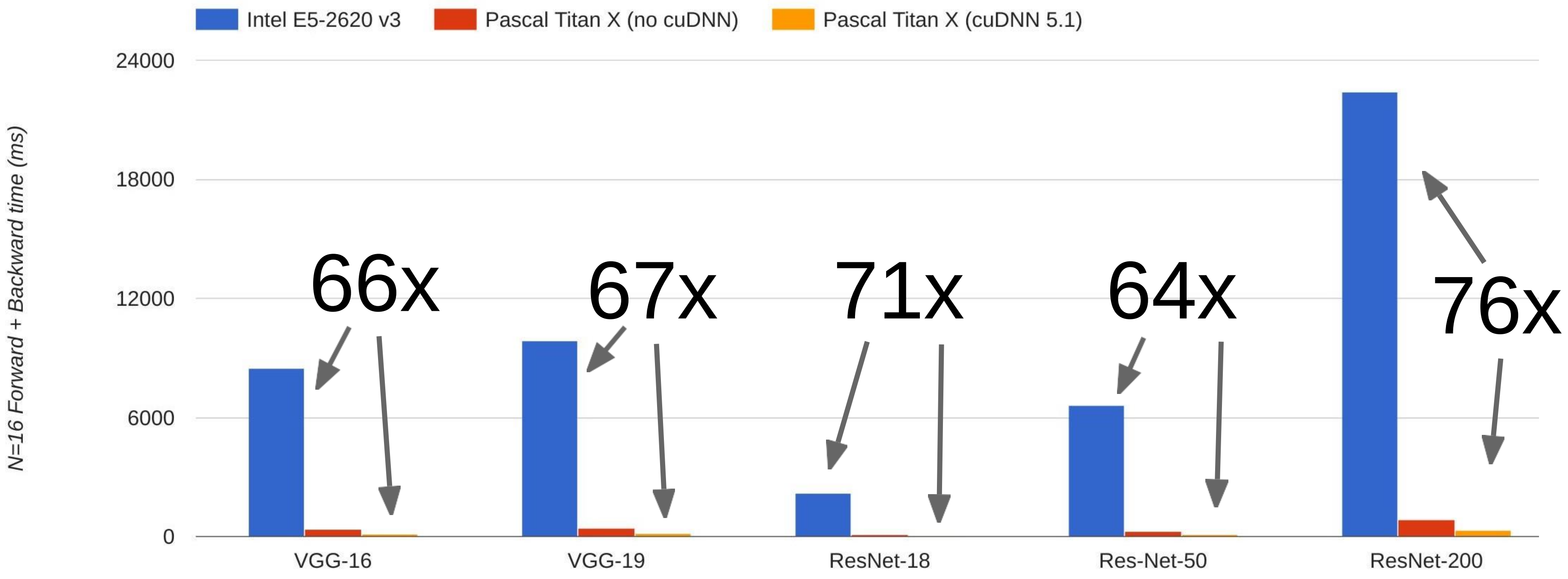
CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

GPU: More cores, but each core is much slower and “dumber”; great for parallel tasks

TPU: Specialized hardware for deep learning

CPU vs GPU in practice

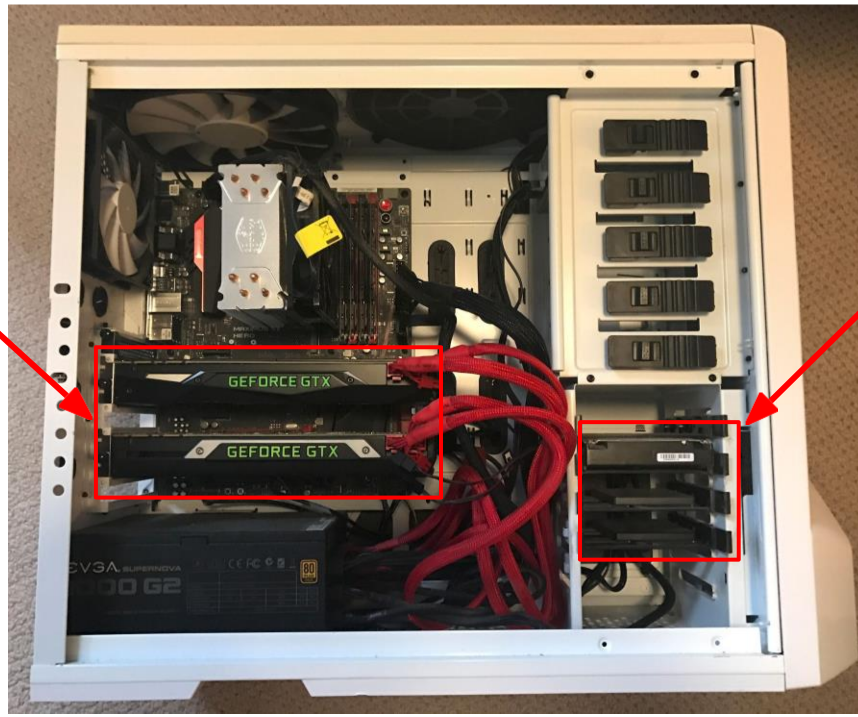
(CPU performance not well-optimized, a little unfair)



Data from https://github.com/jcjohnson/cnn-benchmarks

CPU / GPU Communication

Model
is here



Data is here

If you aren't careful, training can bottleneck on reading data and transferring to GPU!

Solutions:

- Read all data into RAM
- Use SSD instead of HDD
- Use multiple CPU threads to prefetch data

Training: Best practices

- Center (subtract mean from) your data
- To initialize weights, use “Xavier or He initialization”
- Use RELU or leaky RELU or ELU, don’t use sigmoid
- Use mini-batch
- Use data augmentation
- Use regularization
- Use batch normalization
- Use cross-validation for your parameters
- Learning rate: too high? Too low?