

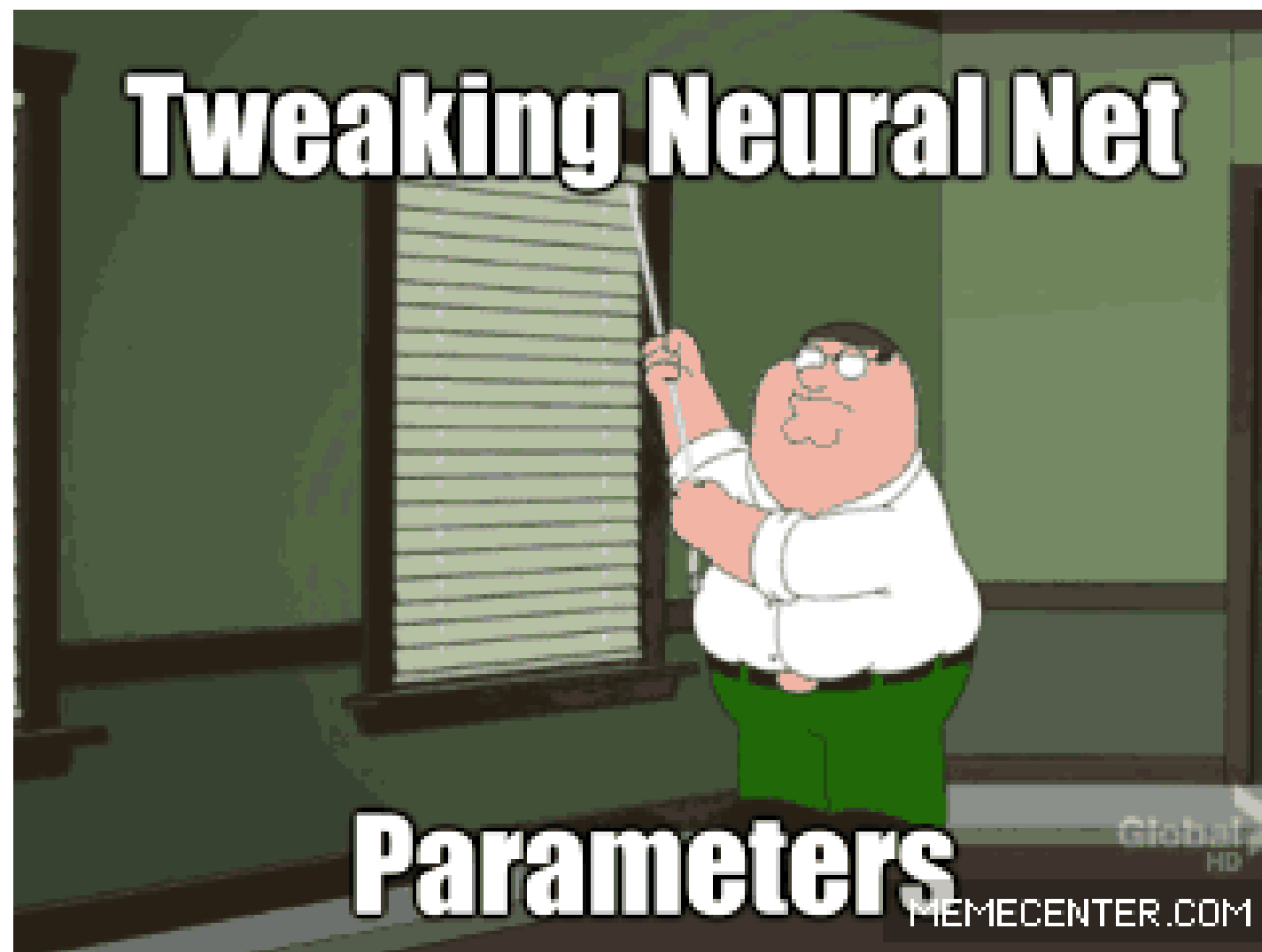
A background illustration of Hiccup and Toothless from the movie 'How to Train Your Dragon' flying over a lush green forest. Hiccup is perched on Toothless's back, pointing forward. Toothless is a large black dragon with yellow eyes. A small white dragon, likely a baby, is flying to the right. The scene is set in a bright, sunny environment with tall trees and a clear sky.

How to Train ~~Your Dragon~~ ConvNets

Stephen Baek

Factors Affecting Your Training Result:

- Network architecture
 - How many layers? How many neurons? Kernel size?
 - Skip connection? Inception (network-in-network)?
- Choice of activation functions
 - Sigmoid? Tanh? ReLU? LeakyReLU? PReLU? Others?
- Data preprocessing and sampling (Data distribution)
 - Data mean and standard deviation, minimum, maximum, noise
 - Train-val-test split, batching
 - Data augmentation, synthetic data, etc.
- ConvNet initialization
 - Initial solution for gradient descent (initial neuron weights and biases)
 - Transfer learning
- Network regularization
 - Dropouts, L1/L2 regularizers, early stopping, multi-task learning, ensemble, etc.
- ... and so many other factors to consider

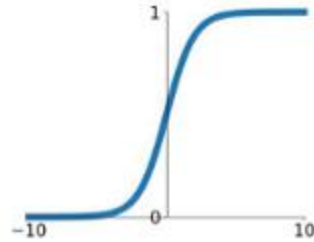


Activation Functions

Recap: Activation Functions

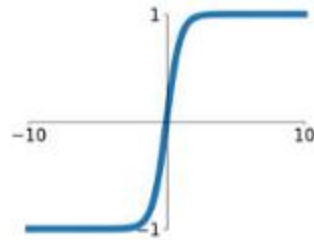
Sigmoid

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



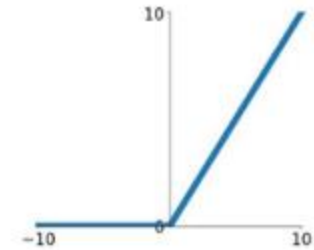
tanh

$$\tanh(x)$$



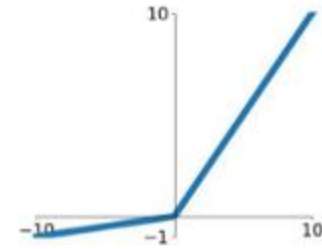
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

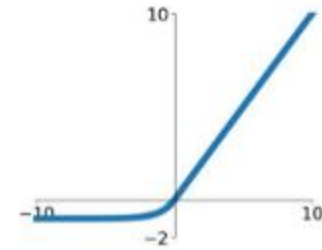


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

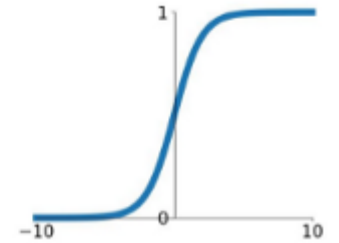


Historically the most popular choice: Sigmoid

- Brain analogy: saturating “firing rate” of a neuron
- Crushes input values to $[0, 1]$.
 - Problems:
 1. Kills off gradients when saturated.
 2. Outputs are always non-negative (not zero-centered).
 3. Exponential! (computationally expensive)

Sigmoid

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

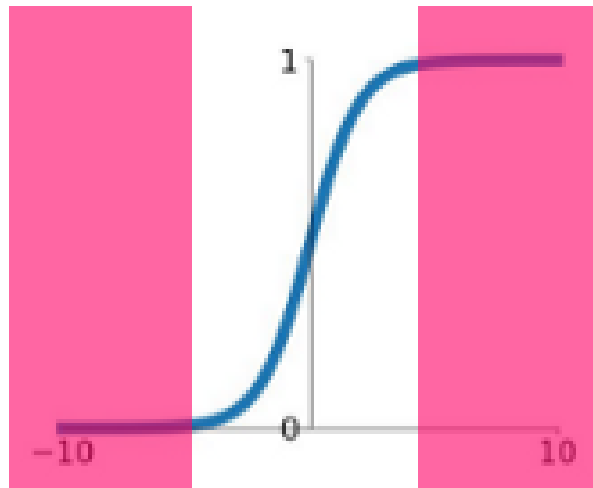
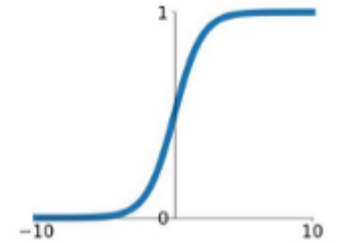


Historically the most popular choice: Sigmoid

- Brain analogy: saturating “firing rate” of a neuron
- Crushes input values to $[0, 1]$.
 - Problems:
 1. Kills off gradients when saturated.
 2. Outputs are always non-negative (not zero-centered).
 3. Exponential! (computationally expensive)

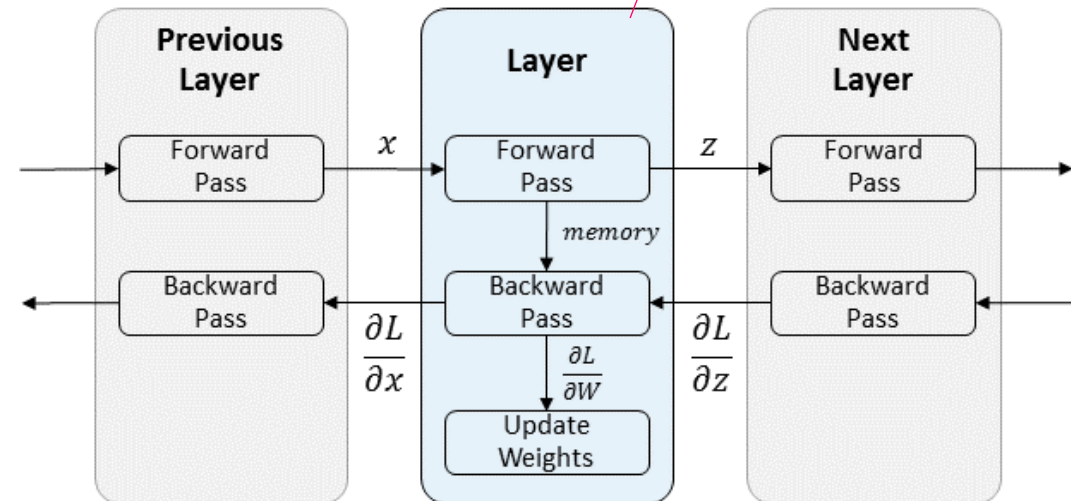
Sigmoid

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



$$\frac{\partial \sigma}{\partial x} = 0$$

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial \sigma} \frac{\partial \sigma}{\partial x}$$

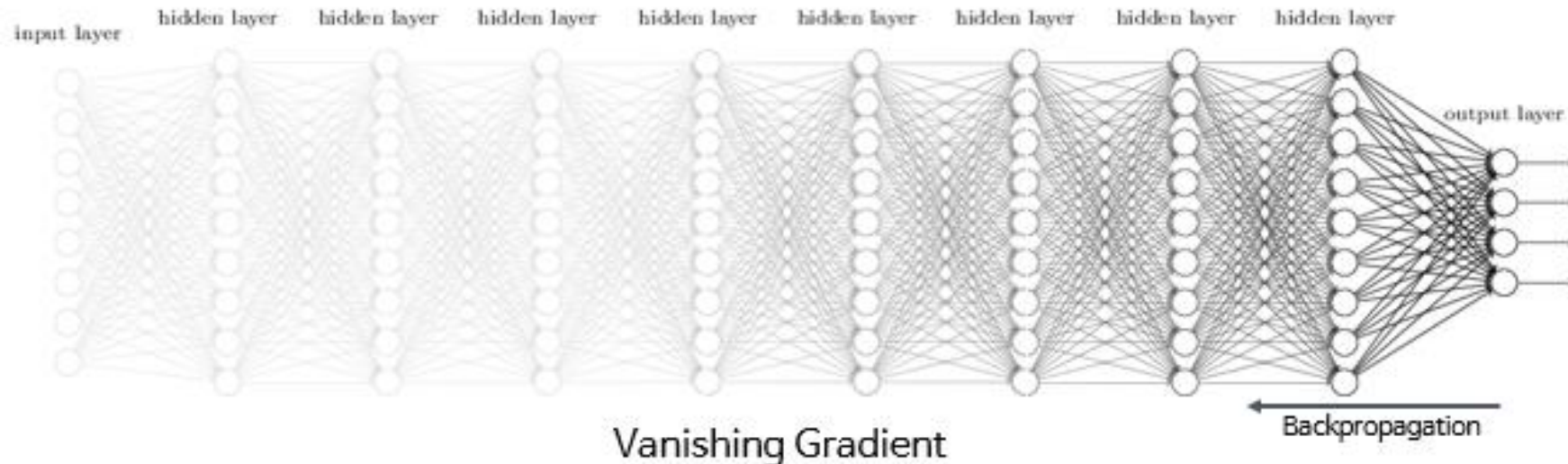
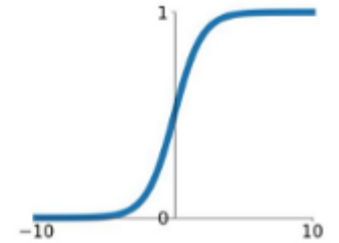


Historically the most popular choice: Sigmoid

- Brain analogy: saturating “firing rate” of a neuron
- Crushes input values to $[0, 1]$.
 - Problems:
 1. Kills off gradients when saturated.
 2. Outputs are always non-negative (not zero-centered).
 3. Exponential! (computationally expensive)

Sigmoid

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

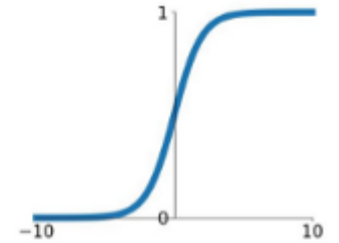


Historically the most popular choice: Sigmoid

- Brain analogy: saturating “firing rate” of a neuron
- Crushes input values to $[0, 1]$.
 - Problems:
 1. Kills off gradients when saturated.
 2. Outputs are always non-negative (not zero-centered).
 3. Exponential! (computationally expensive)

Sigmoid

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



Q. What happens to the gradient if all the input to a neuron is always non-negative? i.e., $x_j \geq 0, \forall x_j$

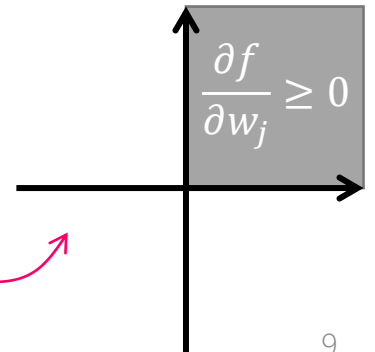
$$f(x | \mathbf{w}, b) = \sum_j w_j x_j + b$$

$$\frac{\partial f}{\partial w_j} = x_j \geq 0 \quad \forall j$$



$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{f}} \frac{\partial \mathbf{f}}{\partial \mathbf{w}}$$

scalar
vector
(gradient)

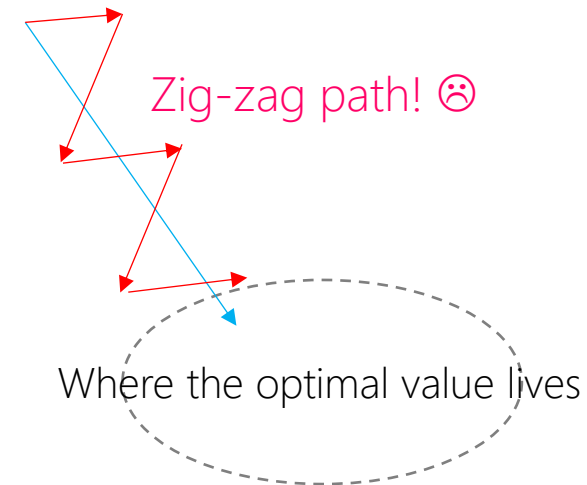
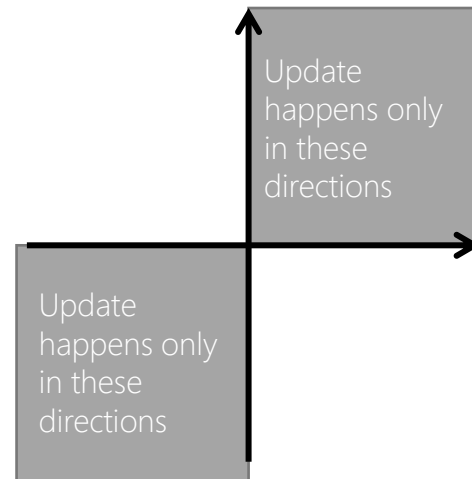
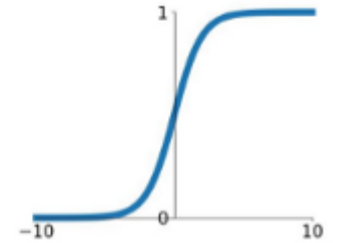


Historically the most popular choice: Sigmoid

- Brain analogy: saturating “firing rate” of a neuron
- Crushes input values to $[0, 1]$.
 - Problems:
 1. Kills off gradients when saturated.
 2. Outputs are always non-negative (not zero-centered).
 3. Exponential! (computationally expensive)

Sigmoid

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

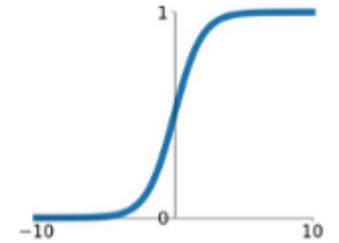


Historically the most popular choice: Sigmoid

- Brain analogy: saturating “firing rate” of a neuron
- Crushes input values to $[0, 1]$.
 - Problems:
 1. Kills off gradients when saturated.
 2. Outputs are always non-negative (not zero-centered).
 3. Exponential! (computationally expensive)

Sigmoid

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



Operation	Input	Output	Algorithm	Complexity
Addition	Two n -digit numbers N, N	One $n+1$ -digit number	Schoolbook addition with carry	$\Theta(n), \Theta(\log(N))$
Subtraction	Two n -digit numbers N, N	One $n+1$ -digit number	Schoolbook subtraction with borrow	$\Theta(n), \Theta(\log(N))$
Multiplication	Two n -digit numbers	One $2n$ -digit number	Schoolbook long multiplication	$O(n^2)$
			Karatsuba algorithm	$O(n^{1.585})$
			3-way Toom–Cook multiplication	$O(n^{1.465})$
			k -way Toom–Cook multiplication	$O(n^{\log(2k-1)/\log k})$
			Mixed-level Toom–Cook (Knuth 4.3.3-T) ^[2]	$O(n^{2^{\sqrt{2} \log n} \log n})$
			Schönhage–Strassen algorithm	$O(n \log n \log \log n)$
			Fürer's algorithm ^[3]	$O(n \log n 2^{O(\log^3 n)})$
Division	Two n -digit numbers	One n -digit number	Schoolbook long division	$O(n^2)$
			Newton–Raphson division	$O(M(n))$
Square root	One n -digit number	One n -digit number	Newton's method	$O(M(n))$
Modular exponentiation	Two n -digit numbers and a k -bit exponent	One n -digit number	Repeated multiplication and reduction	$O(M(n) 2^k)$
			Exponentiation by squaring	$O(M(n) k)$
			Exponentiation with Montgomery reduction	$O(M(n) k)$

Hyperbolic Tangent (LeCun et al. 1991)

- “Rescaled sigmoid”

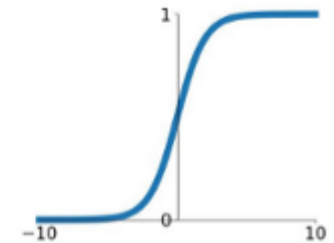
$$2 \frac{1}{1 + e^{-x}} - \frac{(1 + e^{-x})}{1 + e^{-x}} = \frac{1 - e^{-x}}{1 + e^{-x}}$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

- Now it's zero-centered 😊
 - but still...
 - Kills off gradients when saturated.
 - Exponential! (computationally expensive)

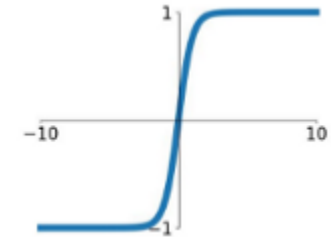
Sigmoid

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



tanh

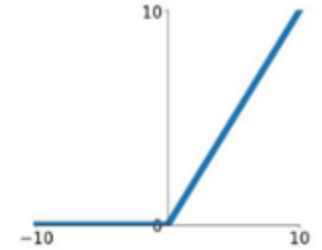
$$\tanh(x)$$



Rectified Linear Unit (Krizhevsky et al., 2012)

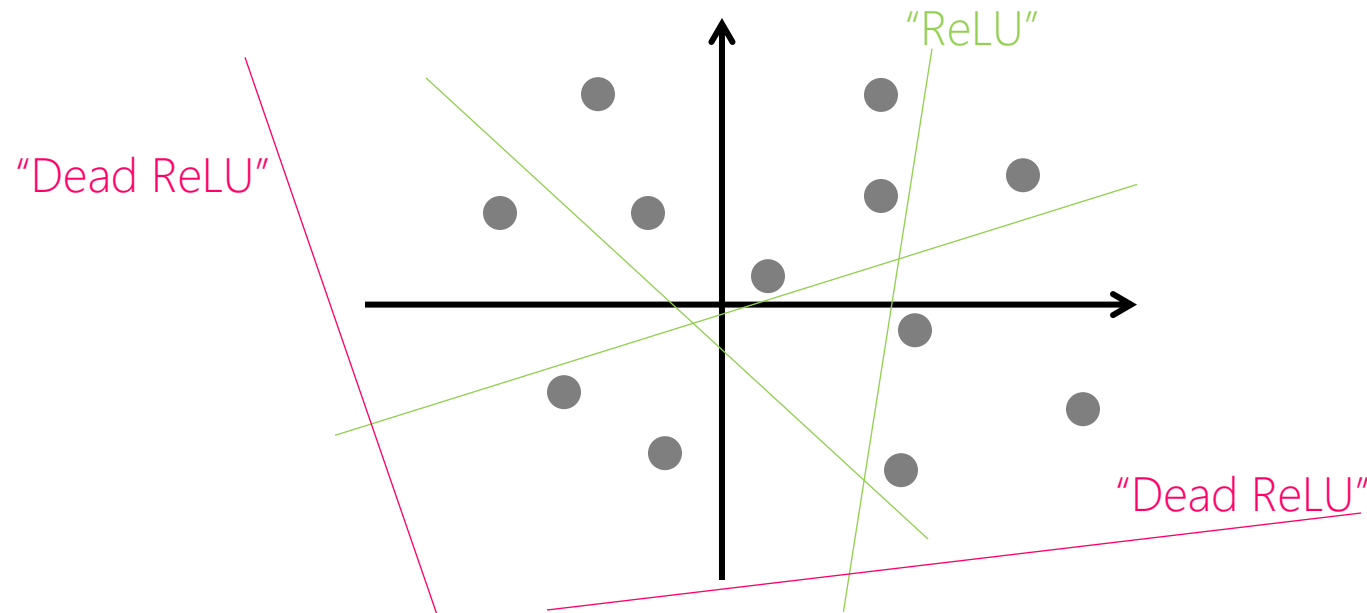
- No saturation when $x > 0$ 😊
- Computationally super efficient. 😊
- Actually, biologically more plausible (modern neuroscience) 😊
- In practice, converges much faster than sigmoid or tanh. 😊
- Not zero-centered though... ☹️
- Zero gradient when $x < 0 \rightarrow$ “Dead ReLU” neurons ☹️

ReLU
 $\max(0, x)$



Rectified Linear Unit (Krizhevsky et al., 2012)

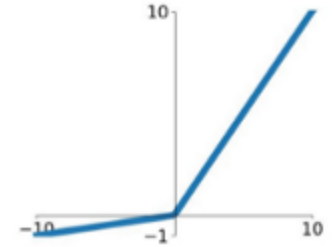
- Dead ReLU
 - A "dead" ReLU always outputs zero for any input.
 - This happens when a large negative bias term is developed.
 - In turn, the dead ReLU neuron will not take any role in the network.
 - "Decision plane" outside the data space.



“Leaky” ReLU (Mass et al., 2013)

- No saturation
- No dead ReLU situation

Leaky ReLU
 $\max(0.1x, x)$



- Leaky ReLU

$$f(x) = \max(\alpha x, x)$$

- α is set by the user
- Parametric Leaky ReLU (PReLU) (He et al., 2015)

$$f(x) = \max(\alpha x, x)$$

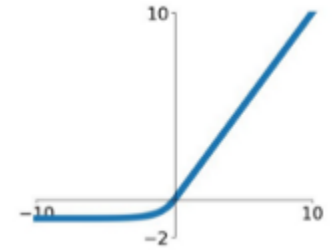
- α is learned from data (backprop)

Exponential Linear Units (Clevert et al., 2015)

- All the benefits of ReLU, plus...
 - Close to zero-mean output
 - Negative saturation → robustness to noise than Leaky ReLUs

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



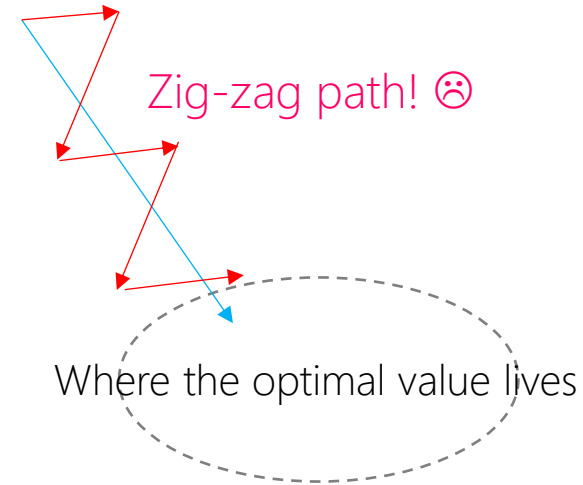
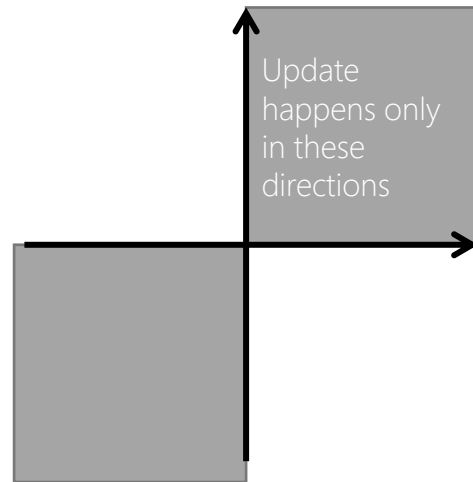
- ... but at expenses of computing exponential... ☹

TL;DR

- “Never” use sigmoid
- tanh is good for some cases.
- ReLU is always the good starting point.
- Try out Leaky ReLU, PReLU, ELU, etc. See if they give any better result.

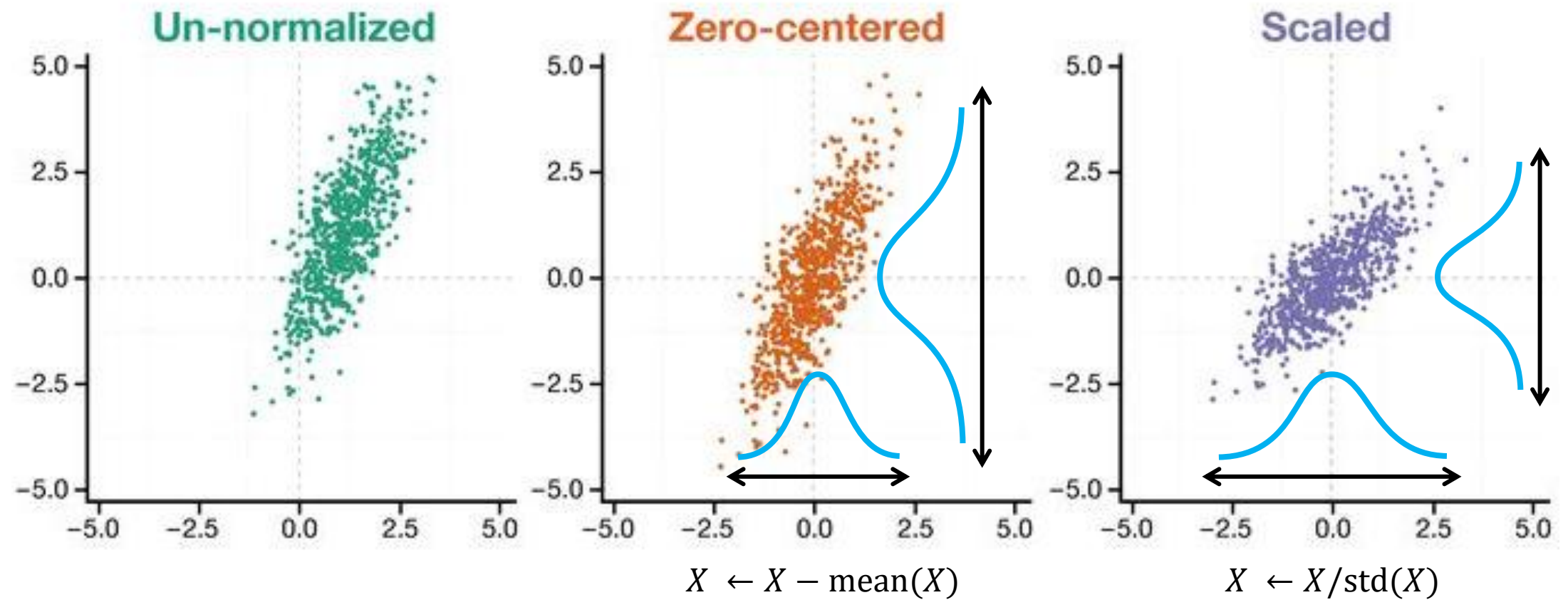
Data Normalization & Whitening

Recall: when inputs to a neuron ≥ 0

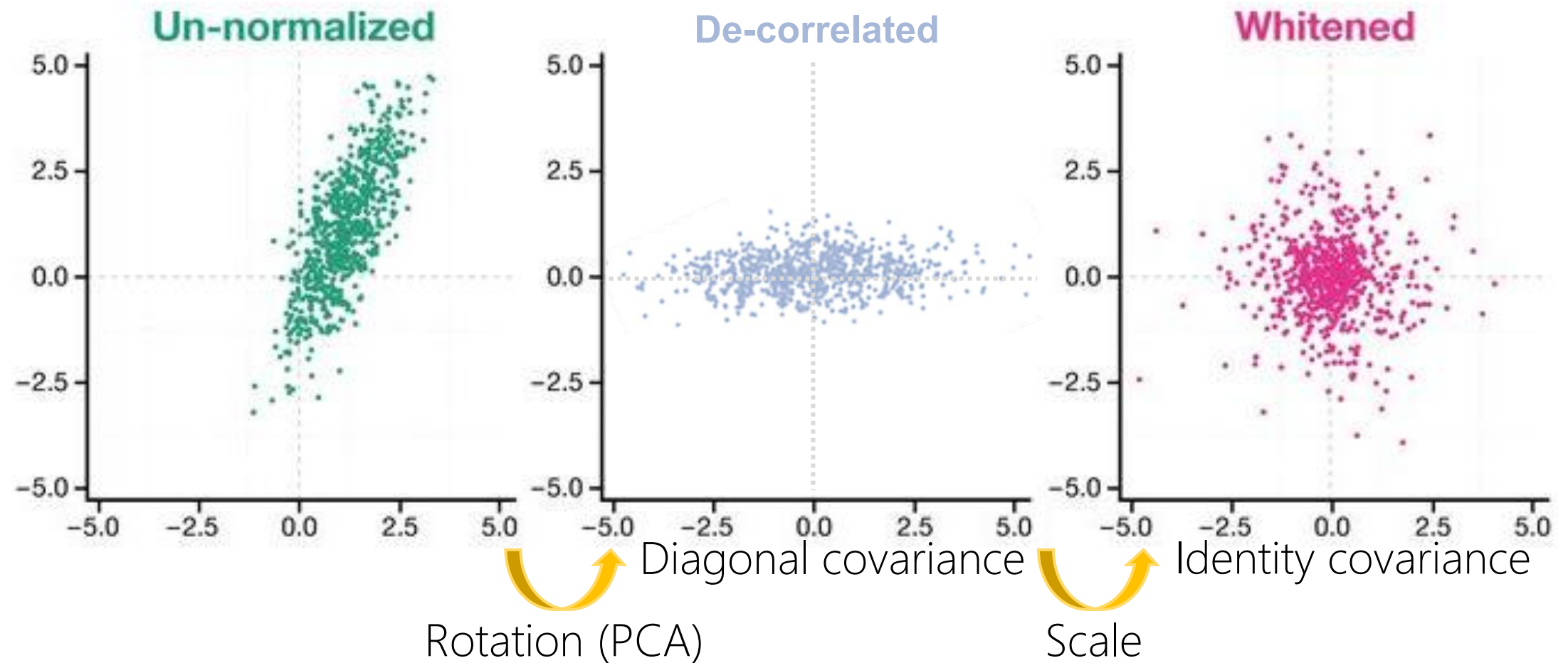


- This also means you need **zero-centered** data!

Standard Normalization



PCA Whitening

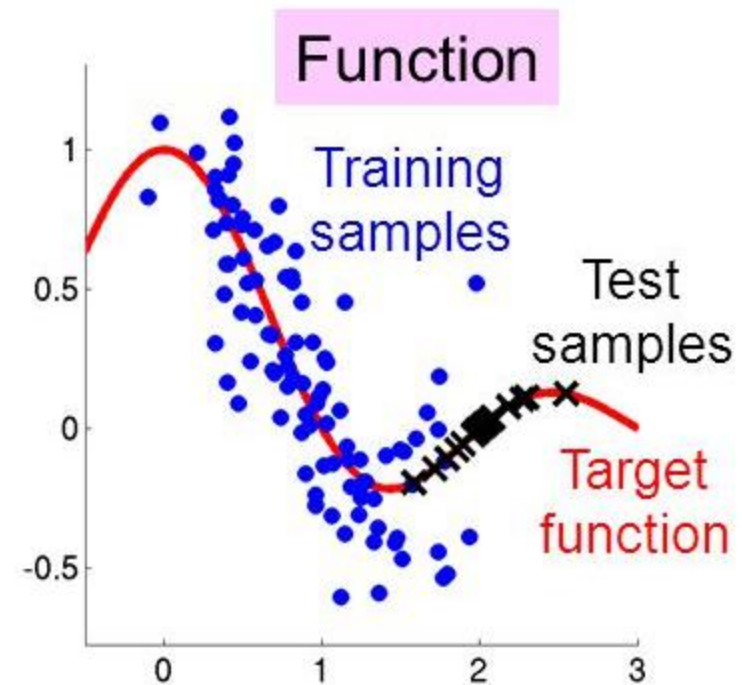
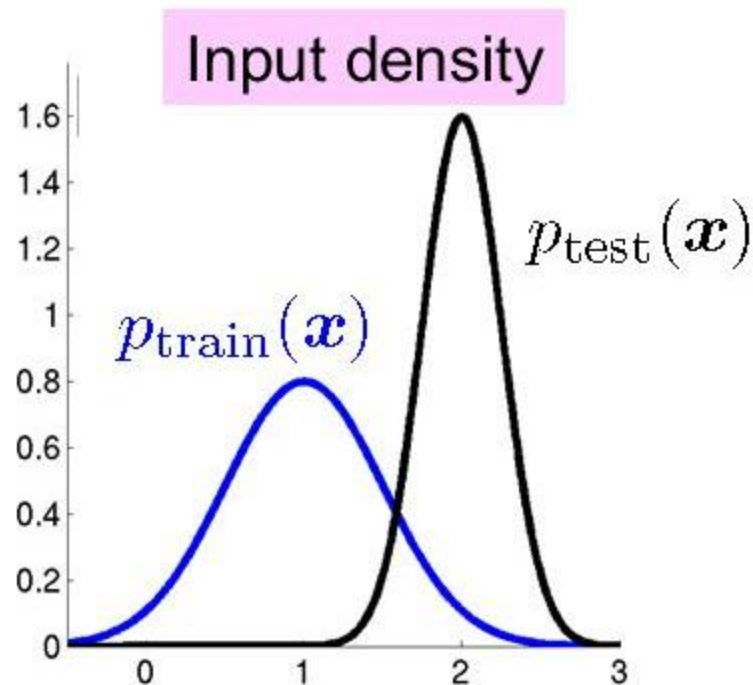


Batch Normalization

- So, we just learned that the input layer can benefit from normalization.
- Why not do the same thing also for the hidden layers?
- Ioffe and Szegedy. (2015) <https://arxiv.org/pdf/1502.03167.pdf>

Covariate Shift

- Test samples have different distribution than training.
- Bit of “extrapolation” is required.



Batch Normalization

- Idea: Improve training by mitigating internal covariate shift.

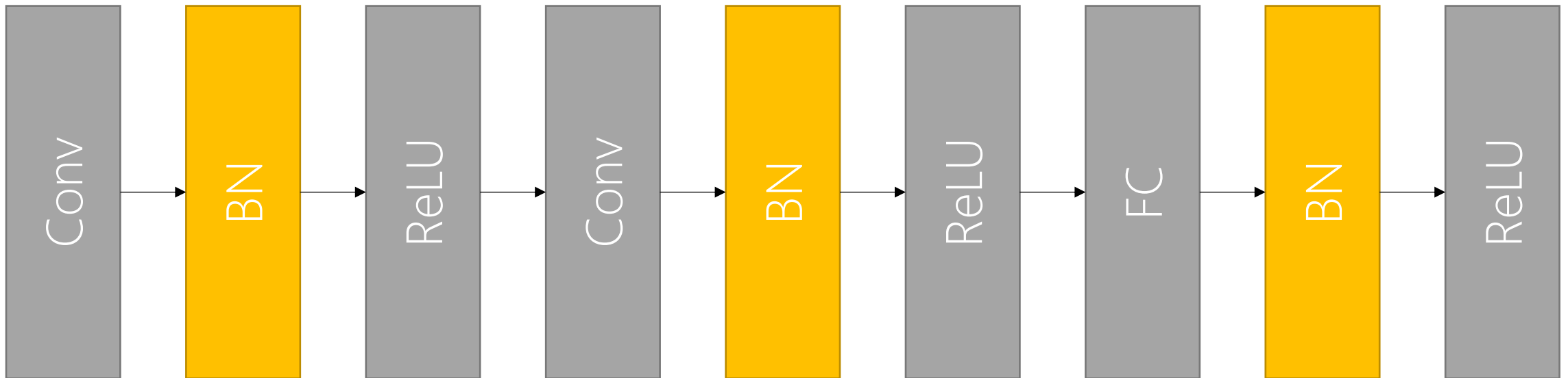
Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batch Normalization

- Batch normalization layers are added typically after FC/Conv and before nonlinearity (activation).



Batch Normalization (Advantages)

- Allows each layer of a network to learn by itself a little bit more **independently of other layers**.
- Allows higher learning rates because BN makes sure that there's no neuron that goes extremely high or extremely low (i.e. **no gradient explosion**).
- Enables stable training at larger batch sizes (Bjorck et al., 2018; De & Smith, 2020)
- Noise statistics among batch → Slight **regularization** effect (Hoffer et al., 2017; Luo et al., 2019).
- **Smoothens** the loss landscape (Santurkar et al., 2018).
- Becomes **less dependent to initialization**

Reading (Optional):

- Bjorck et al. 2018: <https://arxiv.org/pdf/1806.02375.pdf>
- De & Smith 2020: <https://arxiv.org/pdf/2002.10444.pdf>
- Hoffer et al. 2017: <https://arxiv.org/pdf/1705.08741.pdf>
- Luo et al. 2019: <https://arxiv.org/pdf/1809.00846.pdf>
- Santurkar et al. 2018: <https://proceedings.neurips.cc/paper/2018/file/905056c1ac1dad141560467e0a99e1cf-Paper.pdf>

Batch Normalization (Disadvantages)

- Surprisingly **expensive computationally**. Incurs memory overhead (Rota Buló et al., 2018)
- Significantly increases the time required to evaluate the gradient in some networks (Gitman & Ginsburg, 2017)
- Introduces a **discrepancy** between the behavior of the model during training and at inference time (Summers & Dinneen, 2019; Singh & Shrivastava, 2019)

Reading (Optional):

- Rota Buló et al. 2018: https://openaccess.thecvf.com/content_cvpr_2018/papers/Bulo_In-Place_Activated_BatchNorm_CVPR_2018_paper.pdf
- Gitman & Ginsburg 2017: <https://arxiv.org/pdf/1709.08145.pdf>
- Summers & Dinneen 2019: <https://arxiv.org/pdf/1906.03548.pdf>
- Singh & Shrivastava 2019: <https://arxiv.org/pdf/1904.06031.pdf>

Initializing Weights

Weight Initialization

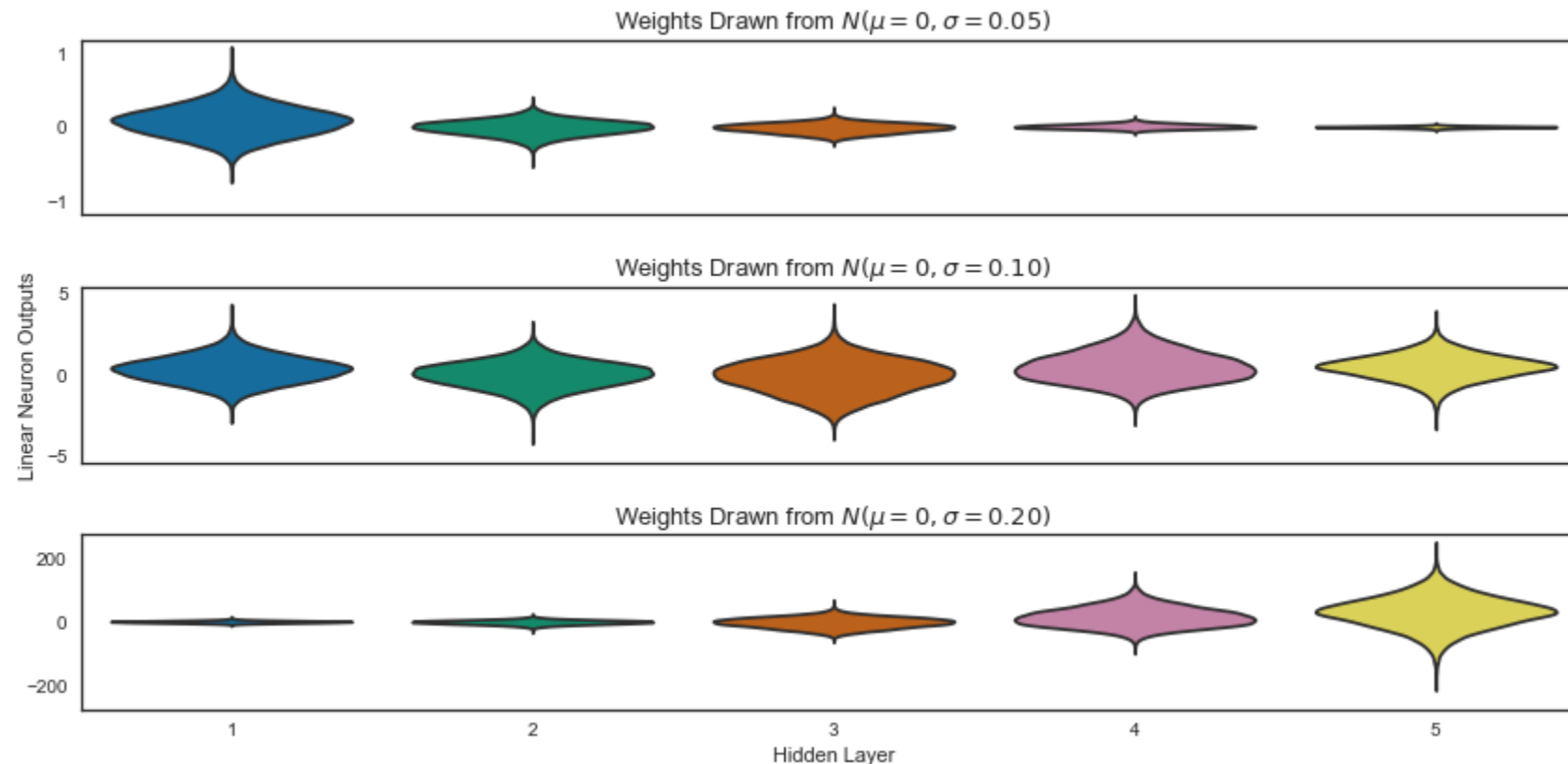
- Simplest way: $W = 0$ (or some constant)
 - What happens in this case? (hint: local gradient direction)
- Weight values **move in the same direction!**
- Never do this!

Random Initialization of Weights

- In practice, neural weights are initialized with some random number.
- Magnitude of the random number matters.
 - Too small → weights dwindle to zero
 - Too big → weights explode
- You need to find an “appropriate” sized random numbers.

Random Initialization of Weights

- Activations of each hidden layer after one forward pass through the network.



How to find the appropriate range?

- Xavier Initialization (Glorot and Bengio 2010): $\text{std}(W_i) = 1/\sqrt{n}$
 - Suppose input X with n components and a linear neuron with random weights W :

$$Y = W_1X_1 + W_2X_2 + \dots + W_nX_n$$

- Then, assuming W_iX_i are uncorrelated (Bienayme formula),

$$\text{Var}(Y) = \text{Var}\left(\sum_i W_iX_i\right) = \sum_i \text{Var}(W_iX_i) = n\text{Var}(W_iX_i)$$

- Meanwhile,
 $\text{Var}(W_iX_i) = E(X_i)^2\text{Var}(W_i) + E(W_i)^2\text{Var}(X_i) + \text{Var}(W_i)\text{Var}(X_i) = \text{Var}(W_i)\text{Var}(X_i)$
- which gives...

$$\text{Var}(Y) = n\text{Var}(W_i)\text{Var}(X_i)$$

- Therefore, to constrain $\text{Var}(Y) = \text{Var}(X_i)$, we need $n\text{Var}(W_i) = 1$

$$\begin{aligned} \text{Var}(XY) &= E[(XY)^2] - \{E[XY]\}^2 && (\because \text{Var}(A) = E[A^2] - \{E[A]\}^2) \\ &= E[X^2Y^2] - \{E[X]E[Y]\}^2 && (\because X \text{ and } Y \text{ are independent}) \\ &= E[X^2]E[Y^2] - E[X]^2E[Y]^2 \\ &= (\sigma_x^2 + \mu_x^2)(\sigma_y^2 + \mu_y^2) - \mu_x^2\mu_y^2 && (\because E[A^2] = \text{Var}(A) + \{E[A]\}^2) \\ &= \sigma_x^2\sigma_y^2 + \sigma_x^2\mu_y^2 + \sigma_y^2\mu_x^2 + \mu_x^2\mu_y^2 - \mu_x^2\mu_y^2 \\ &= \sigma_x^2\sigma_y^2 + \sigma_x^2\mu_y^2 + \sigma_y^2\mu_x^2 \end{aligned}$$

How to find the appropriate range?

- In case of ReLU, Xavier initialization doesn't hold.
 - Derivation of Xavier initialization: assumes a linear neuron.
 - Works alright for Sigmoid and tanh, but not for "rectifiers"
- To over-simplify, half of the outputs (the negative part) dies out under ReLU
- He et al. (2015): $\text{std}(W_i) = 2/\sqrt{n}$

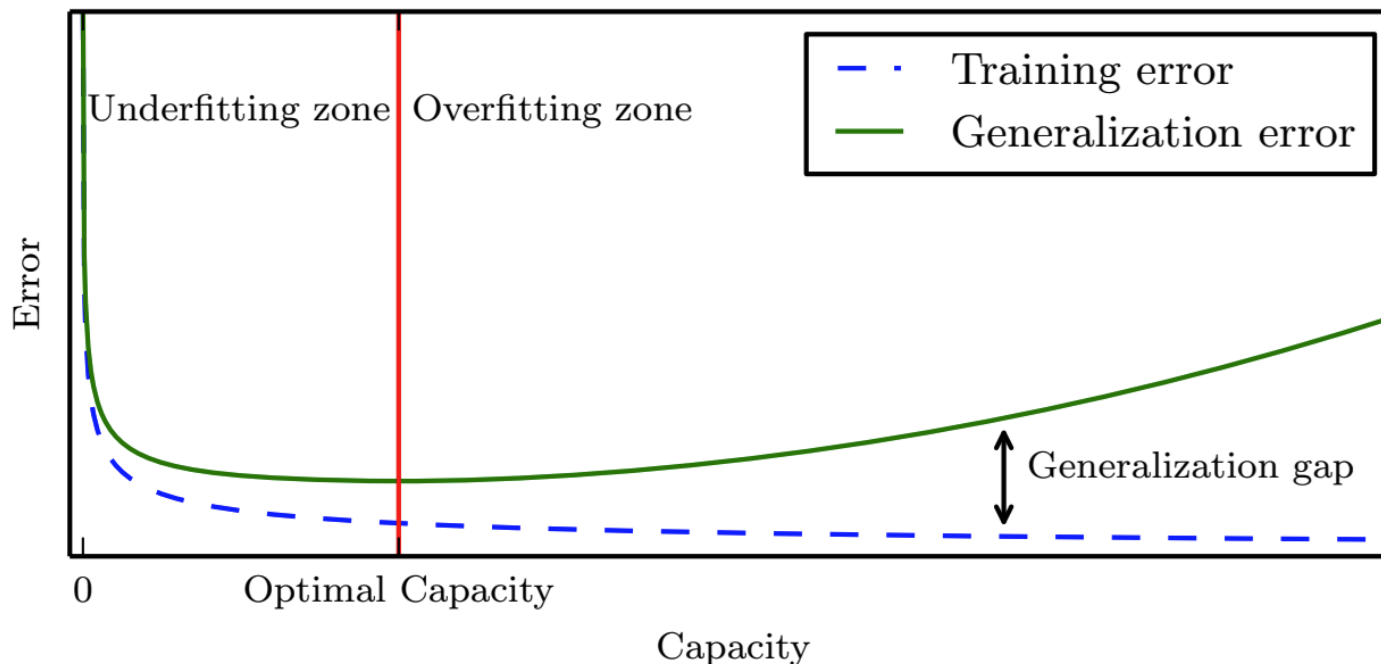
Dropout



**THE BEST WAY TO
EXPLAIN OVERFITTING**

Basic Concepts

- The ability to perform well on *unobserved inputs* is called **generalization**.
- **Underfitting**: when the model is not able to obtain sufficient training error.
- **Overfitting**: when the gap between training and generalization error too large.

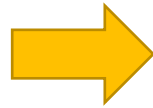


Dropout

- More abstract-level view of overfitting



What network
has seen from
the training set



"Too fluffy, it's not a cat"

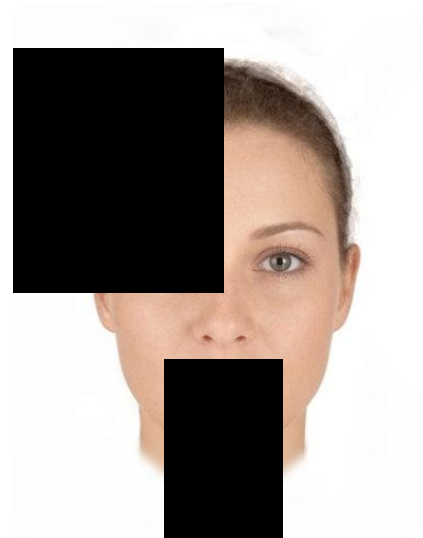


"Too fat. Not a cat"

...

Dropout

- An idea: "let's hide some information while training"
 - Missing information → incentivize generalization of observation



"an eye and a nose
→ face"



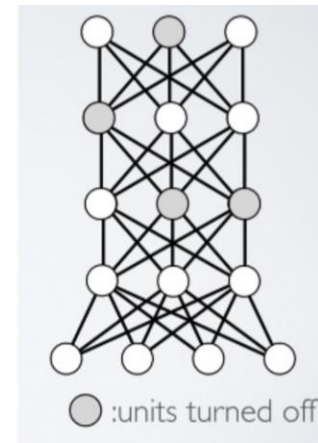
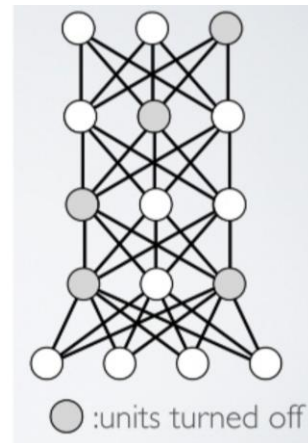
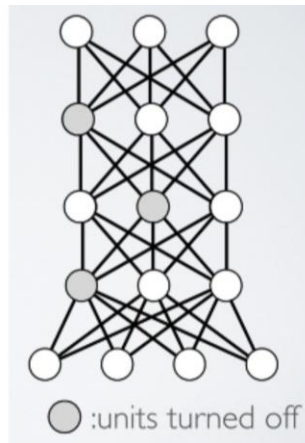
"chin, lips, and nose
→ face"

Human face:

"has to have two eyes, two eyebrows,
one nose with two nostrils, upper and lower lips,
two ears, ..."

Dropout

- But how?
 - Randomly turn off neurons (set inputs to zero) while training



Dropout

- A bonus: increase of training data in a funny way!
 - How many?
 - Example: a FC layer with 4096 units
 - Possible combinations of turning on/off: $2^{4096} \approx 10^{1233}$ cases
 - cf) the total number of atoms in the universe: 10^{82} approx.

Dropout

- Dropout happens only at training time. No dropout during test time.
- Dropout layers can be added to any location in the computational graph.
 - Typically before Conv or FC layers.
- Dropout rate can be controlled independently per each dropout layer.

Data Augmentation

Is this a cat?



How about this?



How about now?



How about now?



How about now?



How about now?



How about now?



How about now?



How about now?



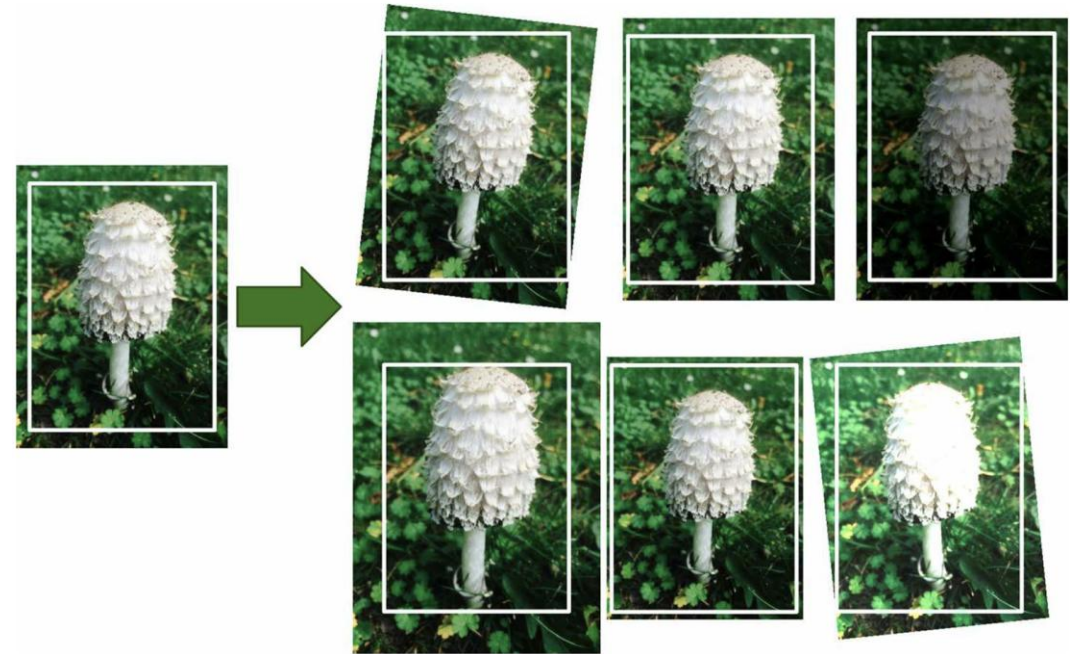
Data Augmentation

- Random rotation, scale, crop, color change in training set
- Even some crazy ideas... because why not?:
 - Stretch
 - Shear
 - Lens distortion
- Point: **Stochasticity!**



Data Augmentation

- Make the model generalize better by training it on **more data**
- If the amount of data available is limited, get around by creating “fake” data and augment them into the training set. How?
- Augmentation has been particularly effective for object recognition. Image transformations (ie. translating, rotating, cropping, brightness correcting, ect) often greatly improve generalization.



Injecting noise and label smoothing

- Injecting (random) noise in the inputs can be a form of data augmentation.
- Injecting noise to output labels? It can be harmful to maximize $\log p(y|x)$ when y is a mistake.
- Prevent this by explicitly model the noise on the label: y is correct with probability $1 - \epsilon$ (epsilon) with small constant ϵ

$$\frac{\epsilon}{k-1} \text{ and } 1 - \epsilon$$

- Label smoothing regularizes a model on a softmax with k output values by replacing hard label 0 and 1 with

Regularization

Preventing weights going crazy...

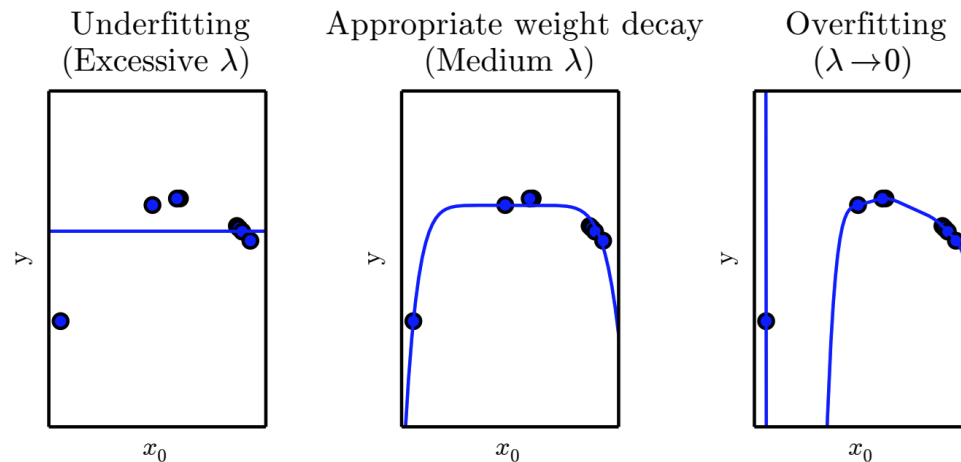
- Too flexible weights → Too flexible models → Overfit!
- You can *regularize* (meaning, prevent the size of weights going too big) by adding a regularization term in the loss function.

$$L(W, b) = \frac{1}{N} \sum_{i=1}^N \|y^{(i)} - f(x^{(i)} | W, b)\|^2 + \lambda R(W, b)$$

Regularization

- **Regularization** is the modification made to a learning algorithm that is intended to reduce its generalization error but not its training error.
- For example, we can modify the loss function for linear regression to include a preference for the weights to have smaller L2 norm as a **regularizer**:

$$J(\mathbf{w}) = \mathcal{L}_{\text{mse}}(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}}) + \lambda \mathbf{w}^{\top} \mathbf{w}$$



Parameter Norm Penalties

- Many regularization approaches are based on limiting the model capacity by adding a parameter norm penalty to the objective (loss) function:

$$J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \underbrace{\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})}_{\text{Data Loss}} + \underbrace{\lambda \Omega(\mathbf{w})}_{\text{Norm Penalty}}$$

where λ is a hyperparameter that controls the relative contribution of the norm penalty term, Ω , relative to the standard data loss function L

- Larger value of λ correspond to more regularization
- Setting λ to 0 results in no regularization
- Norm penalty Ω penalizes only the weights of the affine transformation
- Different choice of Ω can result in different solutions being preferred.

L2 Parameter Regularization

- L2 regularization is aka Ridge Regression or Tikhonov regularization
- The L2 norm penalty commonly known as weight decay

$$J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

$$\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \mathbf{w}$$

To take a single Gradient Descent step to update the weights:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha (\nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \mathbf{w})$$

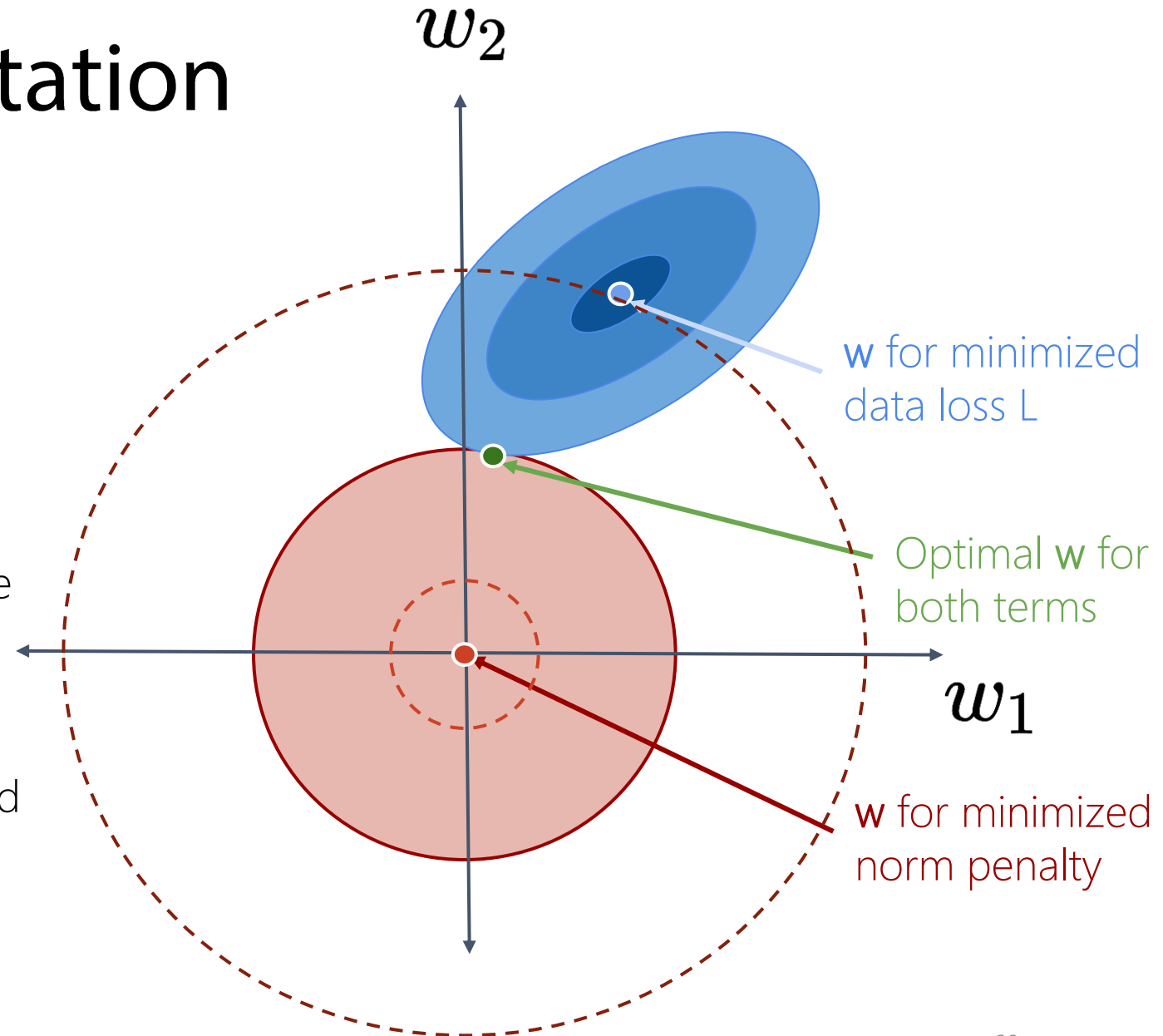
$$\mathbf{w} \leftarrow (1 - \alpha\lambda) \mathbf{w} - \alpha \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$$

< 1

Shrink the weight vector before gradient update

Geometric Interpretation

- L2 regularizations $\|w\|^2$ can be geometrically represented as concentric (red) circles.
- When circle is too small, the params are not useful to the model.
- When the circle region (in red) grows, due to its shape the region intersects the data contour **closer to the origin**, L2 makes both parameters shrink and w_1 near zero.
- When the circle grows too large, you end up with a similar params as data loss.



L1 Parameter Regularization

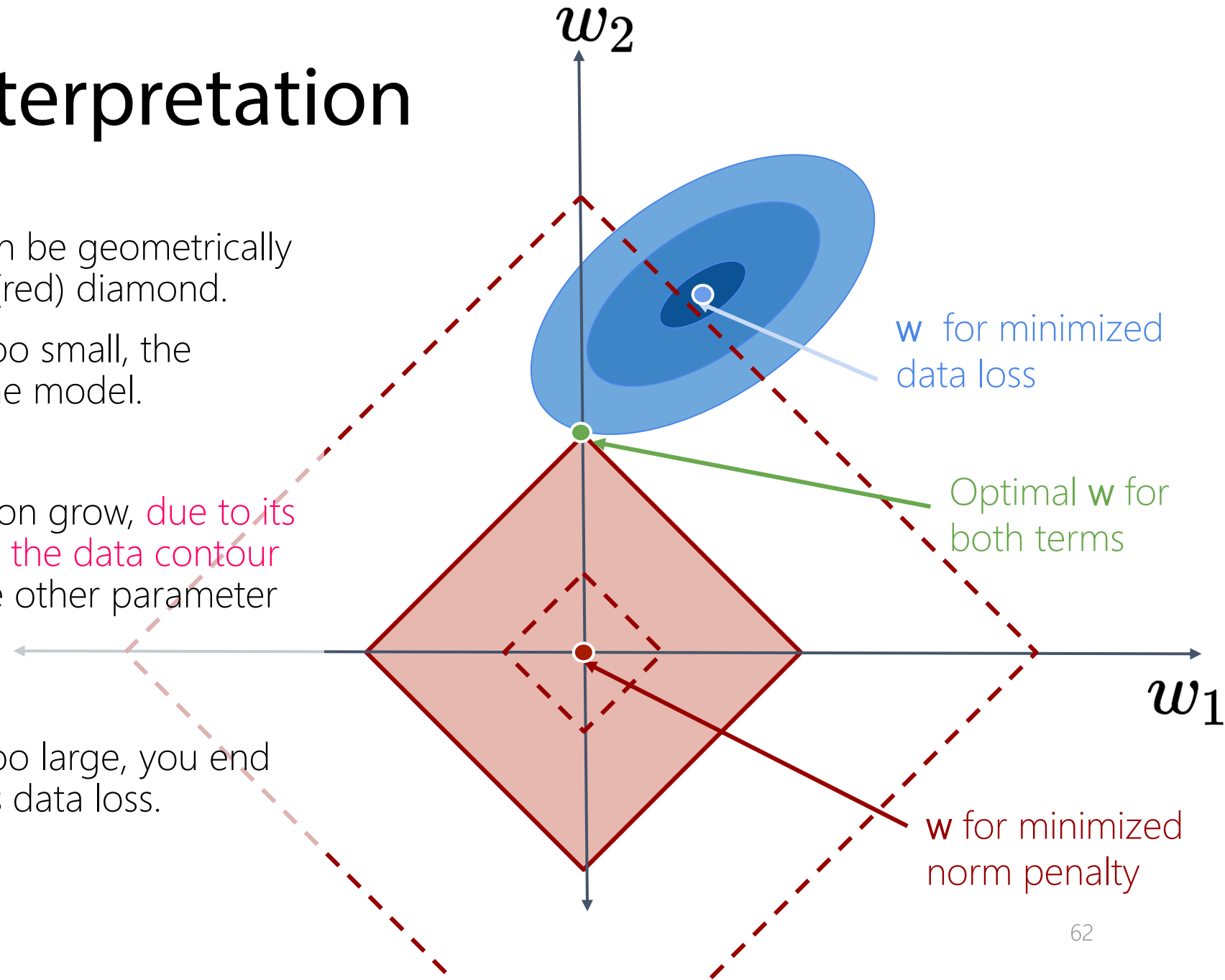
- L1 regularization is aka **LASSO** (least absolute shrinkage and selection operator)
- L1 norm commonly is known as the **Manhattan Distance**.

$$J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \lambda ||\mathbf{w}||_1$$
$$\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \text{sign}(\mathbf{w})$$

- The regularization to the gradient no longer scale linearly with each \mathbf{w} , instead it is a constant factor with a sign equal to $\text{sign}(\mathbf{w})$. Thus, there is **no clean algebraic solution** to approximate J as we have just seen in L2 regularization
- L1 norm makes the parameters to become **sparse** (contains lots of zeros)
- L1 norm tends to cause a subset of the network weights to become zero, suggesting that the corresponding signal may safely be discarded (**dead neuron**).

Geometric Interpretation

- L1 regularizations $\|w\|^1$ can be geometrically represented as concentric (red) diamond.
- When diamond region is too small, the params are not useful to the model.
- When as the diamond region grow, **due to its shape the region intersects the data contour on one of the axis**, thus the other parameter is zero out.
- When diamond region is too large, you end up with a similar params as data loss.

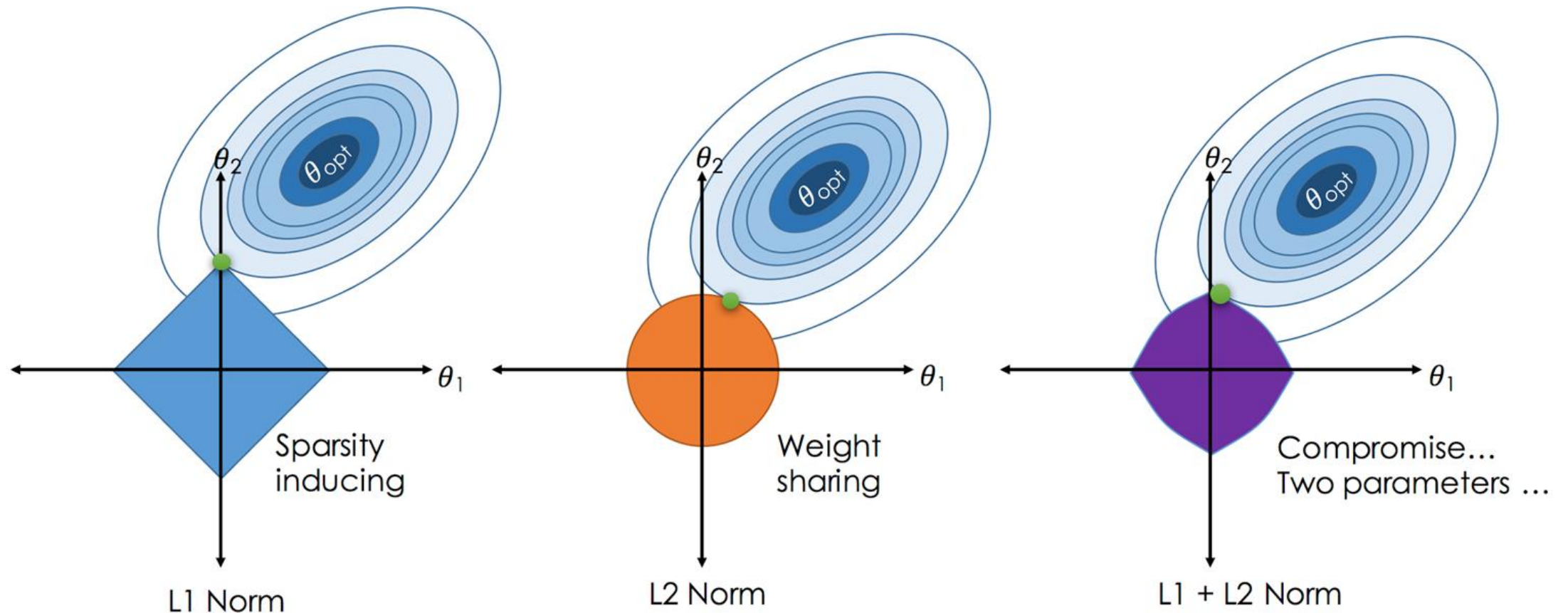


Elastic Net

- Is a middle ground between Ridge (L2) and Lasso (L1) with a **ratio** r .

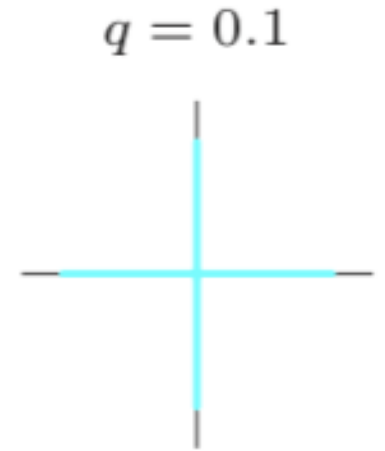
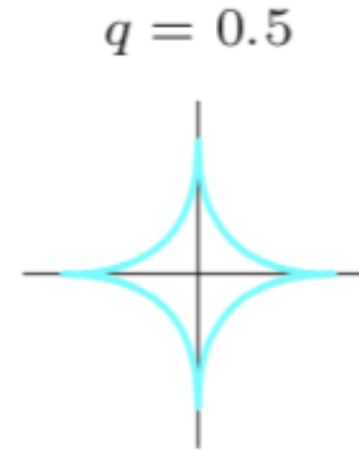
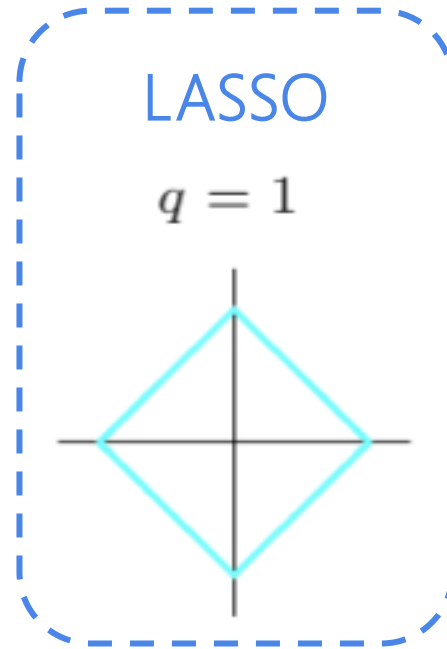
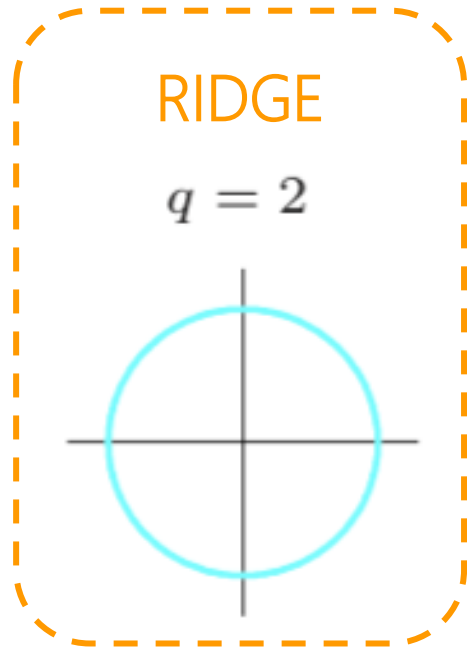
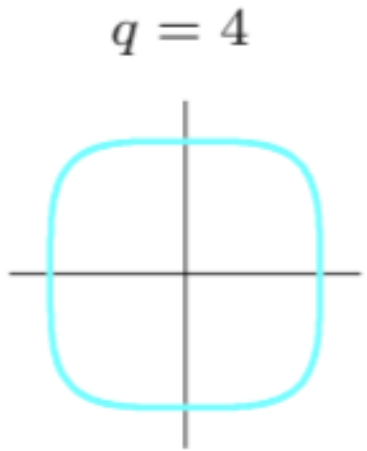
$$J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \underbrace{r\lambda ||\mathbf{w}||_1}_{\text{LASSO}} + \underbrace{(1-r)\lambda ||\mathbf{w}||_2^2}_{\text{RIDGE}}$$

Geometric Interpretation



Family of Parameter Norm Models

$$J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \lambda ||\mathbf{w}||_q$$



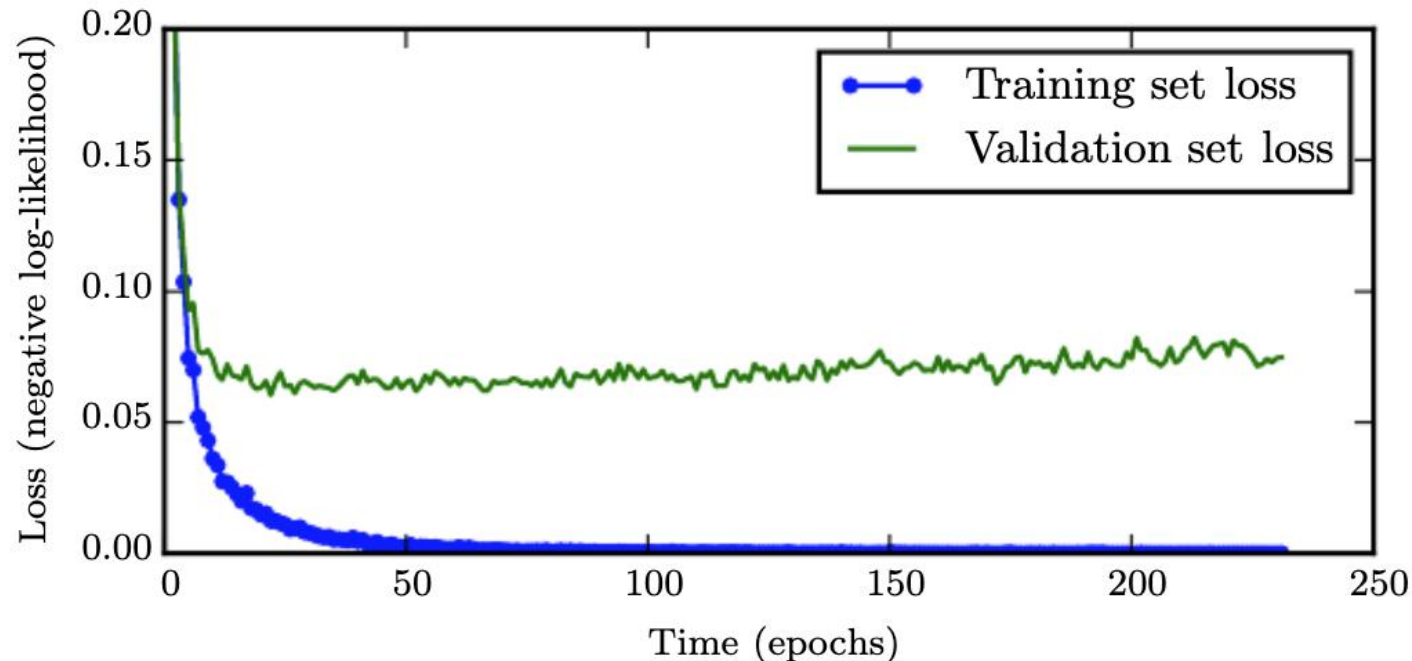
TL;DR

- It is preferable to have at least a little bit of regularization. Ridge (L2) is a good default.
- If you suspect that only a few features are actually useful, use Lasso (L1)
- Lasso may behave erratically when $\# \text{ features} > \# \text{ examples}$ → use Elastic Net

Early Stopping

Early Stopping

- When training large models with sufficient capacity to overfit the task, we often observe reliably that training error decreases steadily over time, but validation error begins to rise again → we can **stop at the lowest validation error!**
- The most commonly used form of regularization in deep learning: simple yet effective!



Early Stopping: Algorithm

Algorithm 7.1 The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

Let n be the number of steps between evaluations.

Let p be the “patience,” the number of times to observe worsening validation set error before giving up.

Let θ_o be the initial parameters.

$\theta \leftarrow \theta_o$

$i \leftarrow 0$

$j \leftarrow 0$

$v \leftarrow \infty$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

while $j < p$ **do**

 Update θ by running the training algorithm for n steps.

$i \leftarrow i + n$

$v' \leftarrow \text{ValidationSetError}(\theta)$

if $v' < v$ **then**

$j \leftarrow 0$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

$v \leftarrow v'$

else

$j \leftarrow j + 1$

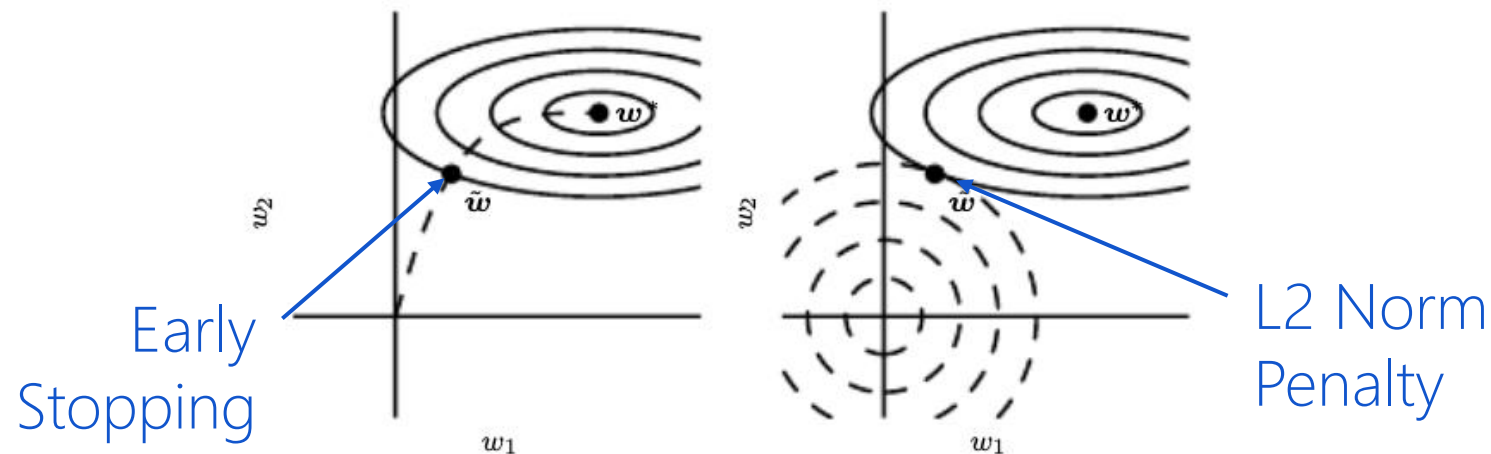
end if

end while

Best parameters are θ^* , best number of training steps is i^* .

Early Stopping: Properties

- Is a very efficient **hyperparameter selection** method (saves the best set).
- Controls the **model capacity** by determining how many steps to fit the training data (and reduce the **computational cost** of the training procedure)
- Unobtrusive form of regularization as almost no change in training procedure
- Can be used either alone or with other regularization strategies
- Is equivalent to L2 regularization, and better because it automatically determines the correct amount of regularization while L2 requires the tuning of hyperparameter λ



Tips on Learning Process & Hyperparameter Tuning

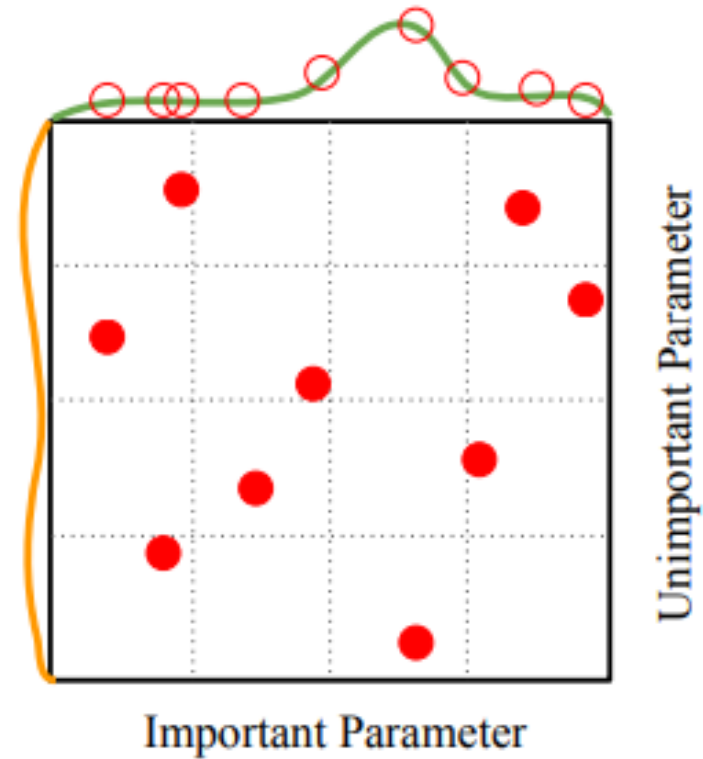
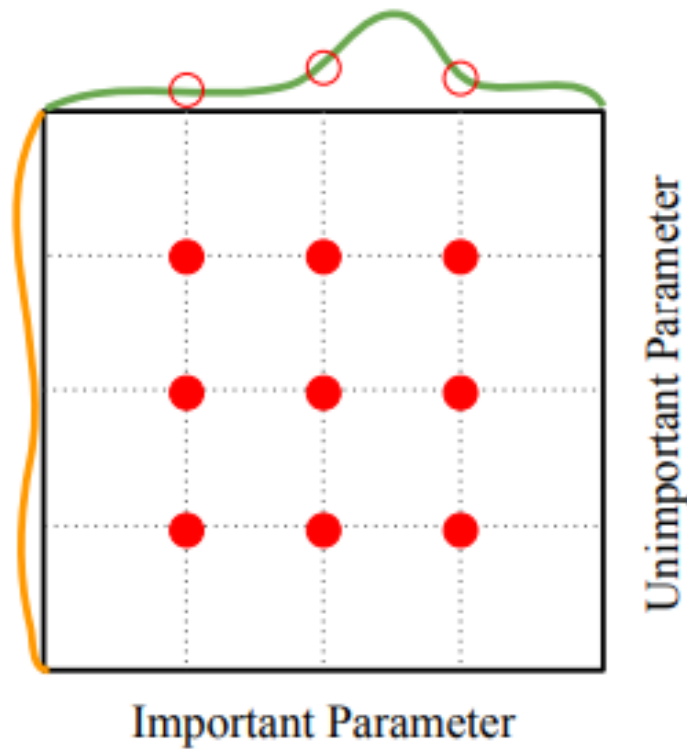
Training Process, Step-by-step

1. Preprocess the data (normalize/whiten, or at least zero-center them)
2. Design the network architecture (start with a simple one if possible)
3. Train with a small amount of data (a subset of your data). Make sure the model can overfit (i.e. loss go down to 0 and accuracy goes 1).
4. Now, the actual training, find the learning rate
 - LR too low: loss won't go down, LR too high: loss will explode.
 - Start with a small number and find the learning rate that makes some observable change.
5. Hyperparameter Tuning!

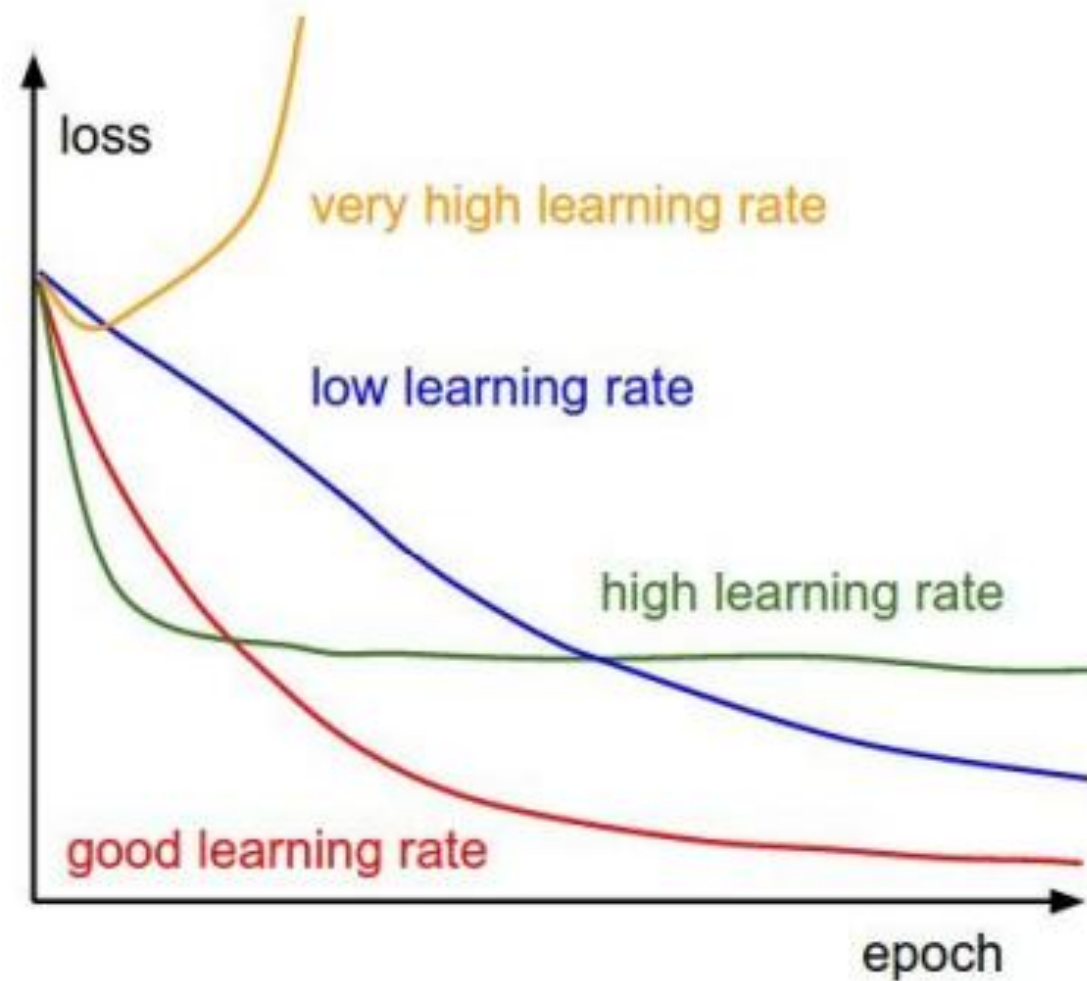
Hyperparameters Tuning

- Major hyperparameters: depth and breadth (or network architecture)
- Others: regularization terms, learning rate and its decay schedule, etc.
- Coarse-to-fine Strategy:
 - First, only a few epochs to get a rough idea of how parameters play
 - Second, longer epochs, do finer search
 - Repeat this as architecture changes.

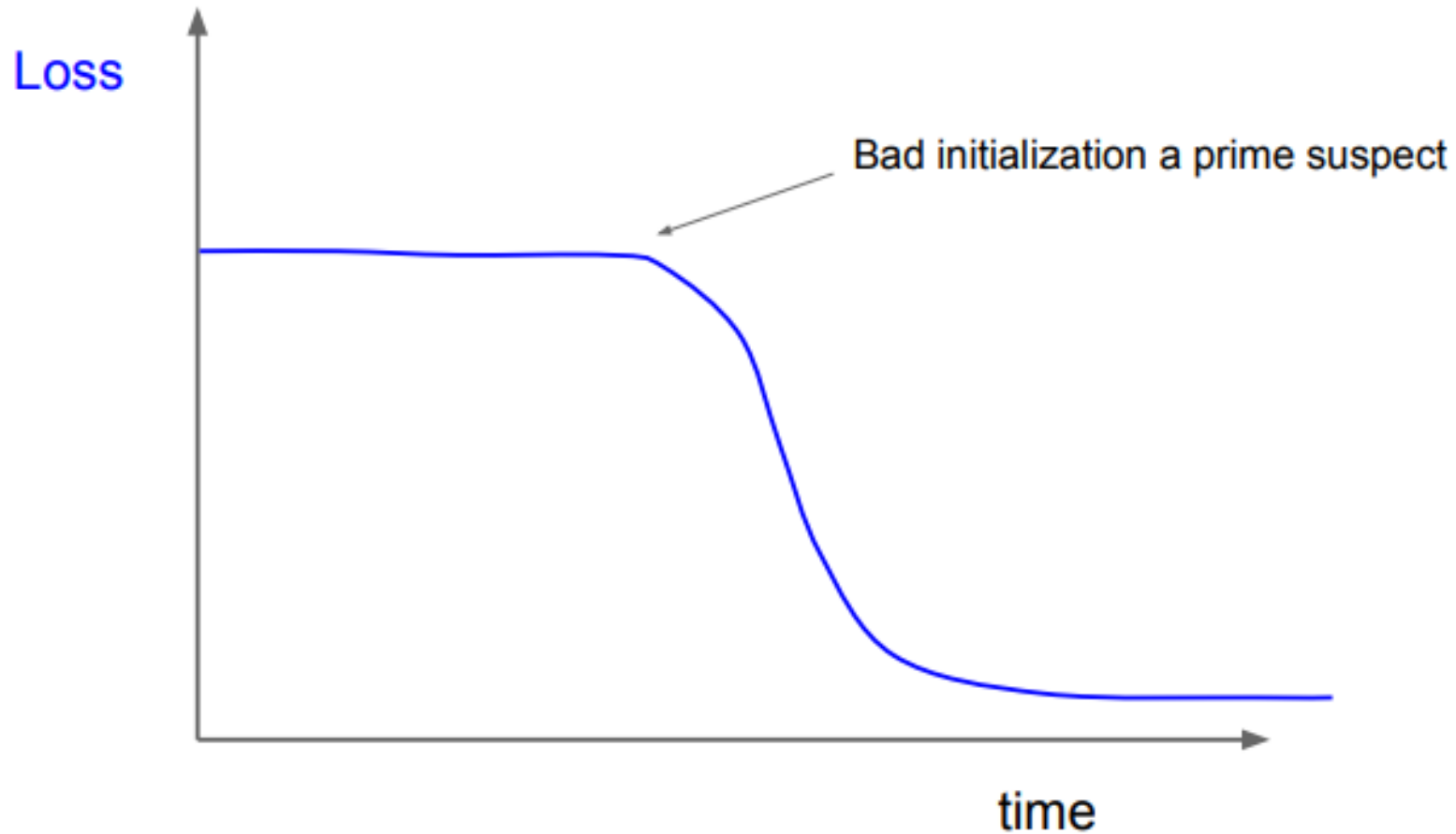
Grid Search vs. Random Search



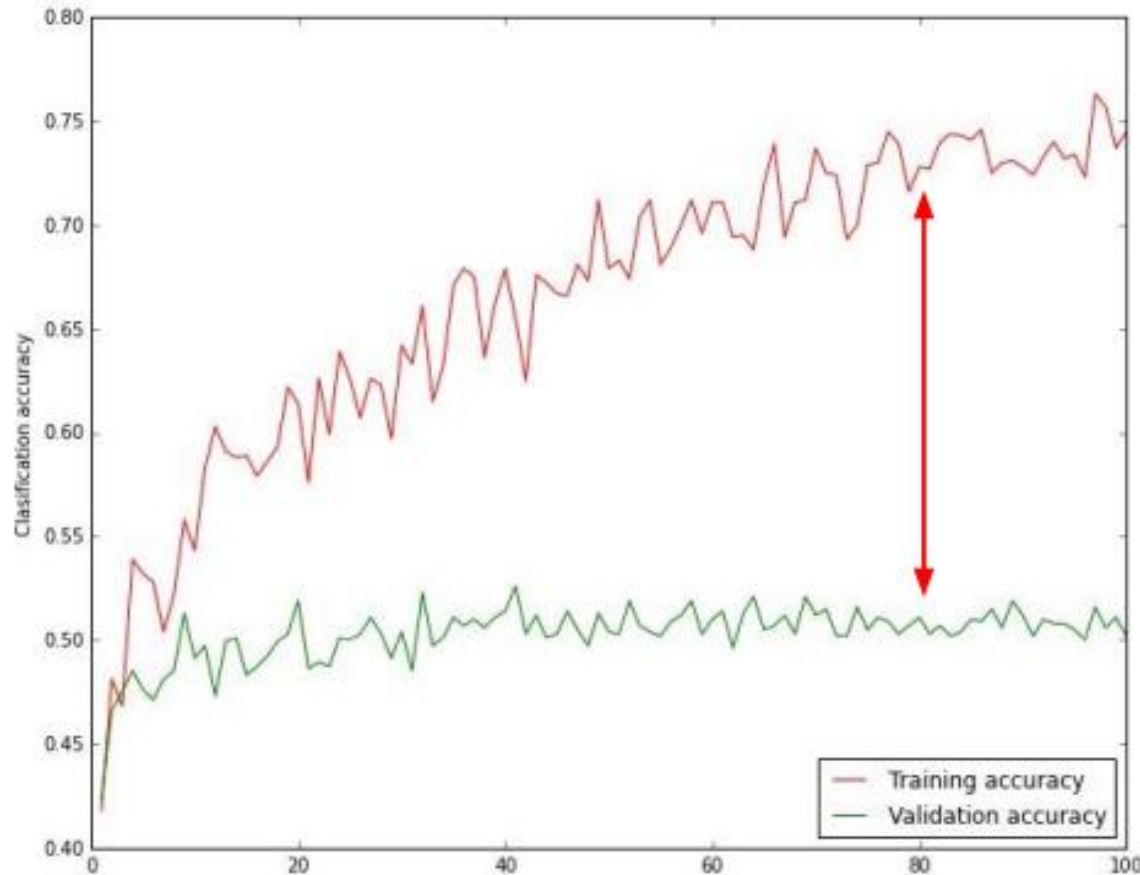
Always monitor and visualize the loss/accuracy!



Always monitor and visualize the loss/accuracy!



Always monitor and visualize the loss/accuracy!



big gap = overfitting
=> increase regularization strength?

no gap
=> increase model capacity?

Tracking the weights also helps

- Percentage of weight update over weight magnitudes
 - Around 0.001 is ideal
 - Around 0.01 is about okay

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

Stanford cs231n