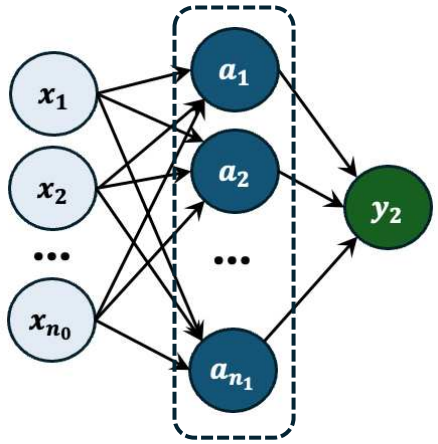
COM SCI M148, Lecture 15:
Regularizing Neural Networks

Prof. Nathan Szymanski
nszymanski@seas.ucla.edu
Department of Materials Science

UCLA

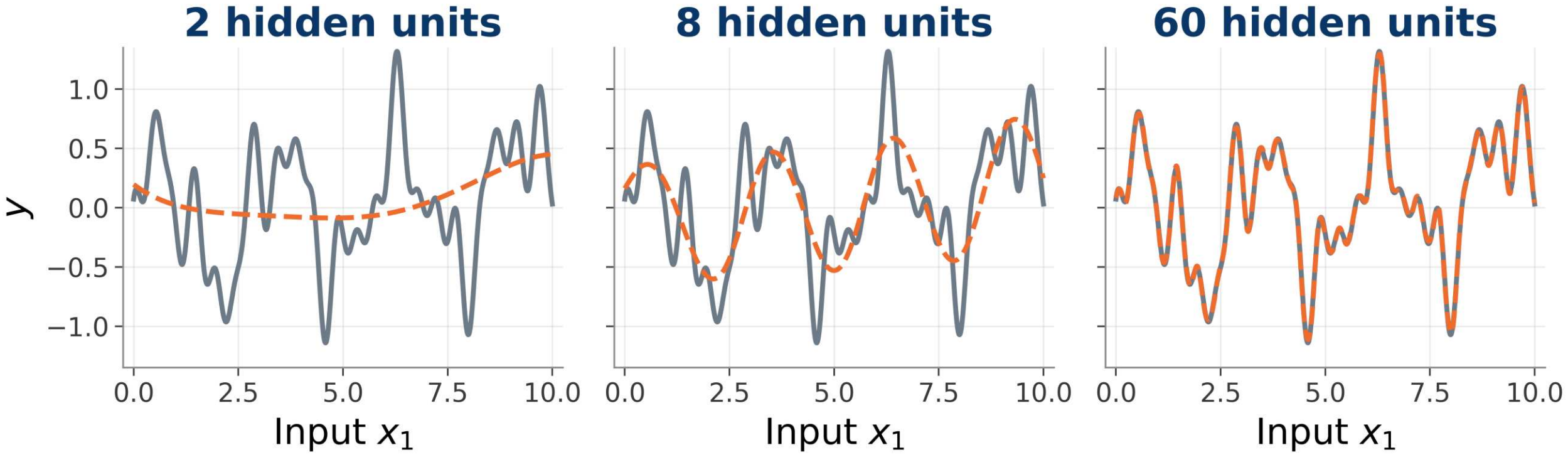
Samueli
School of Engineering

Neural networks: Universal function approximators

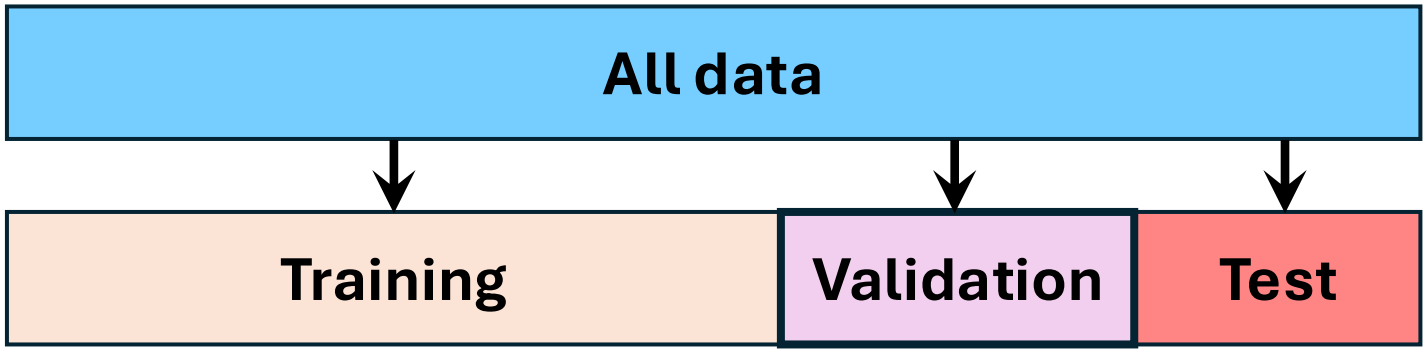


With enough non-linear **hidden units**, we can fit anything!

This makes neural networks great for learning complex relationships, but it also makes them ***very* easy to overfit…**

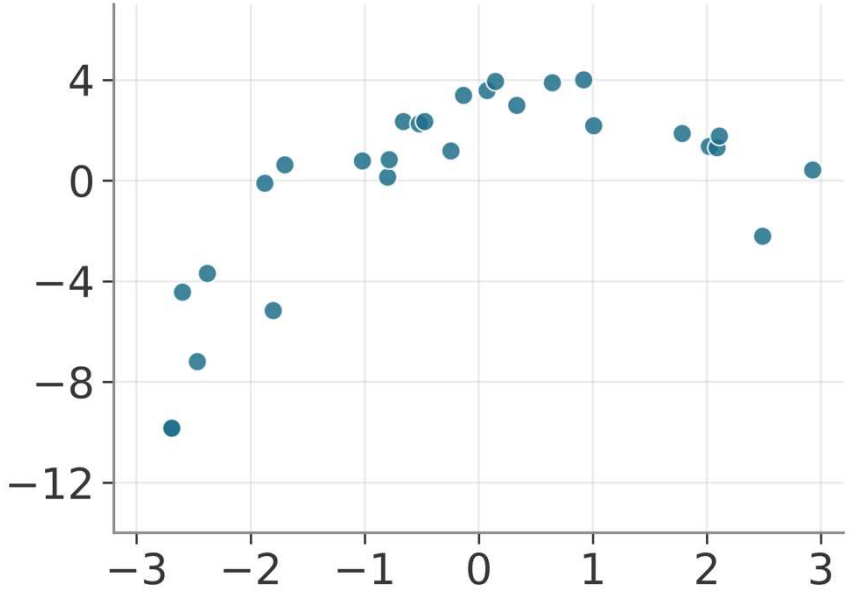


Recall: Use validation to detect overfitting



When we overfit, loss will be low on the training set but high on the validation (and test) set
→ **failure to generalize!**

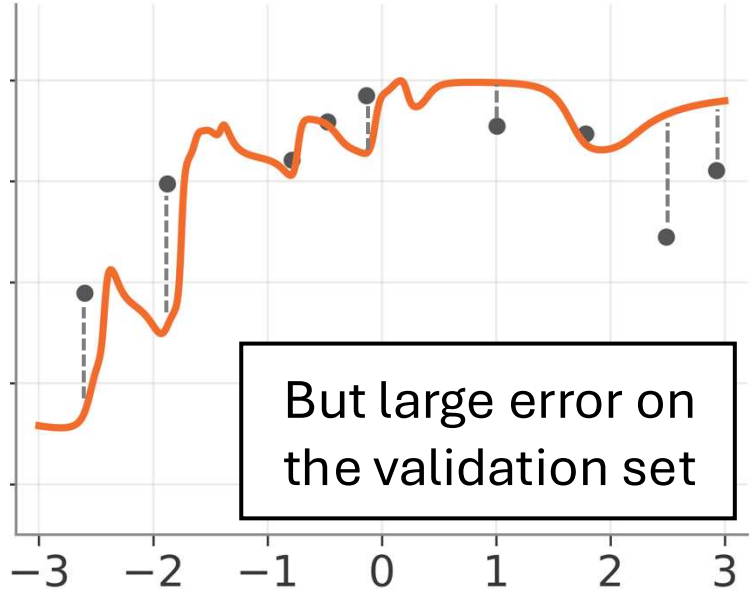
All data



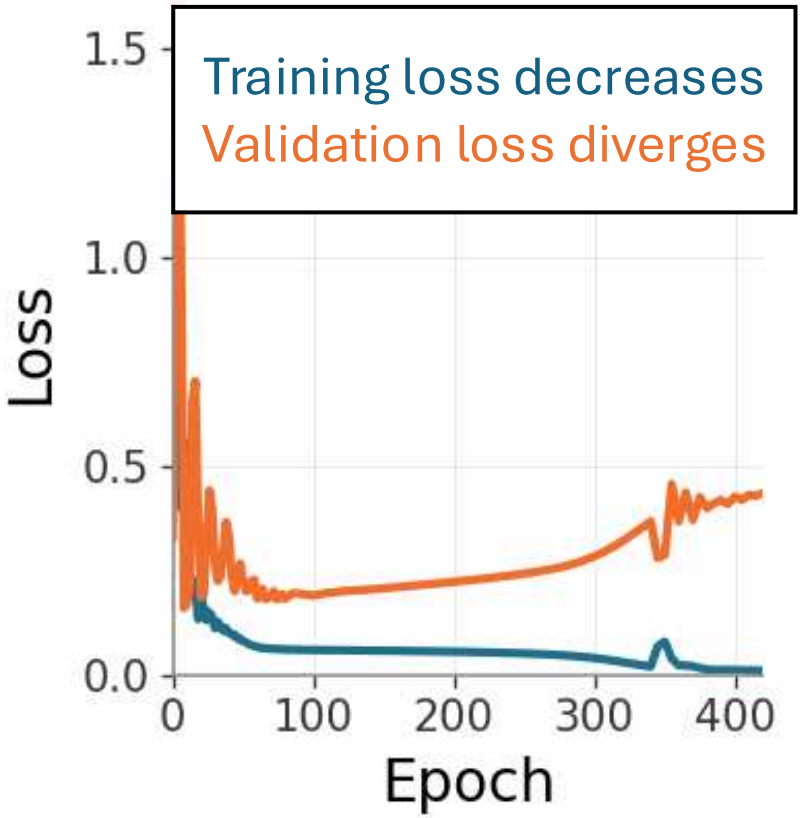
Training set

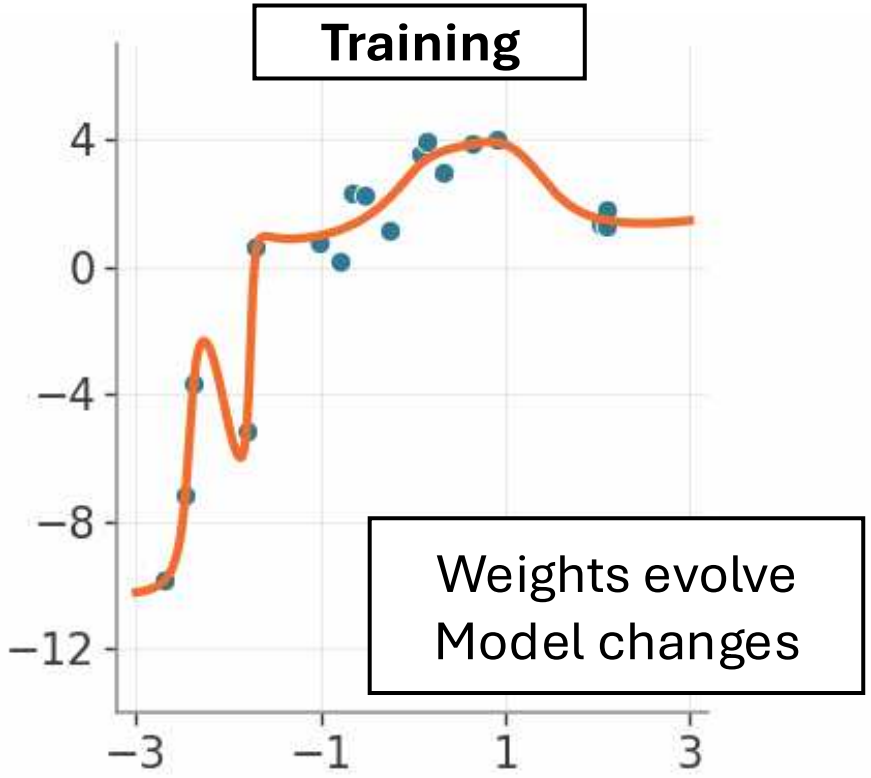


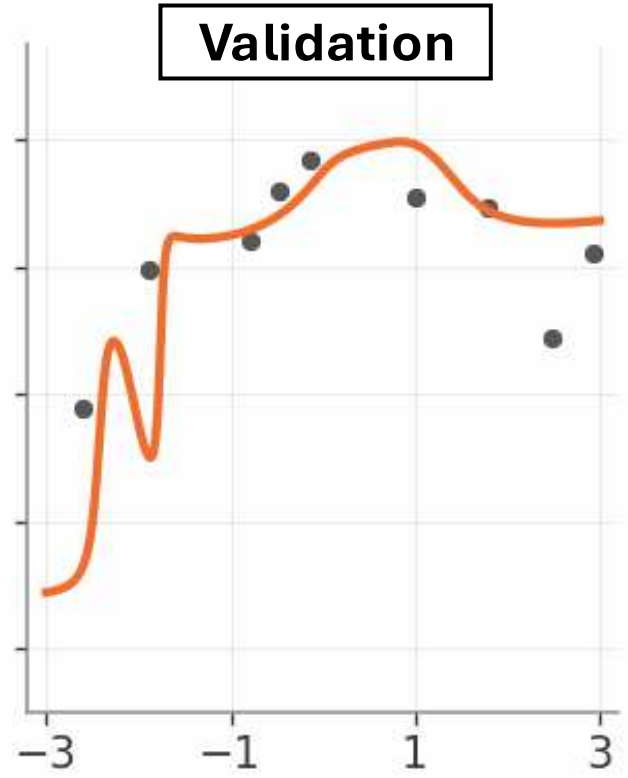
Validation set



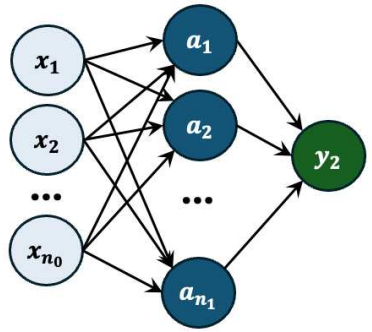
Recall: Loss curve from gradient descent

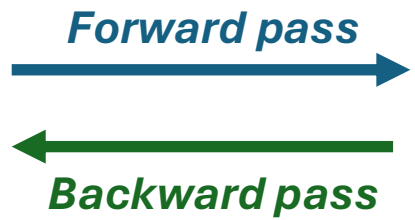




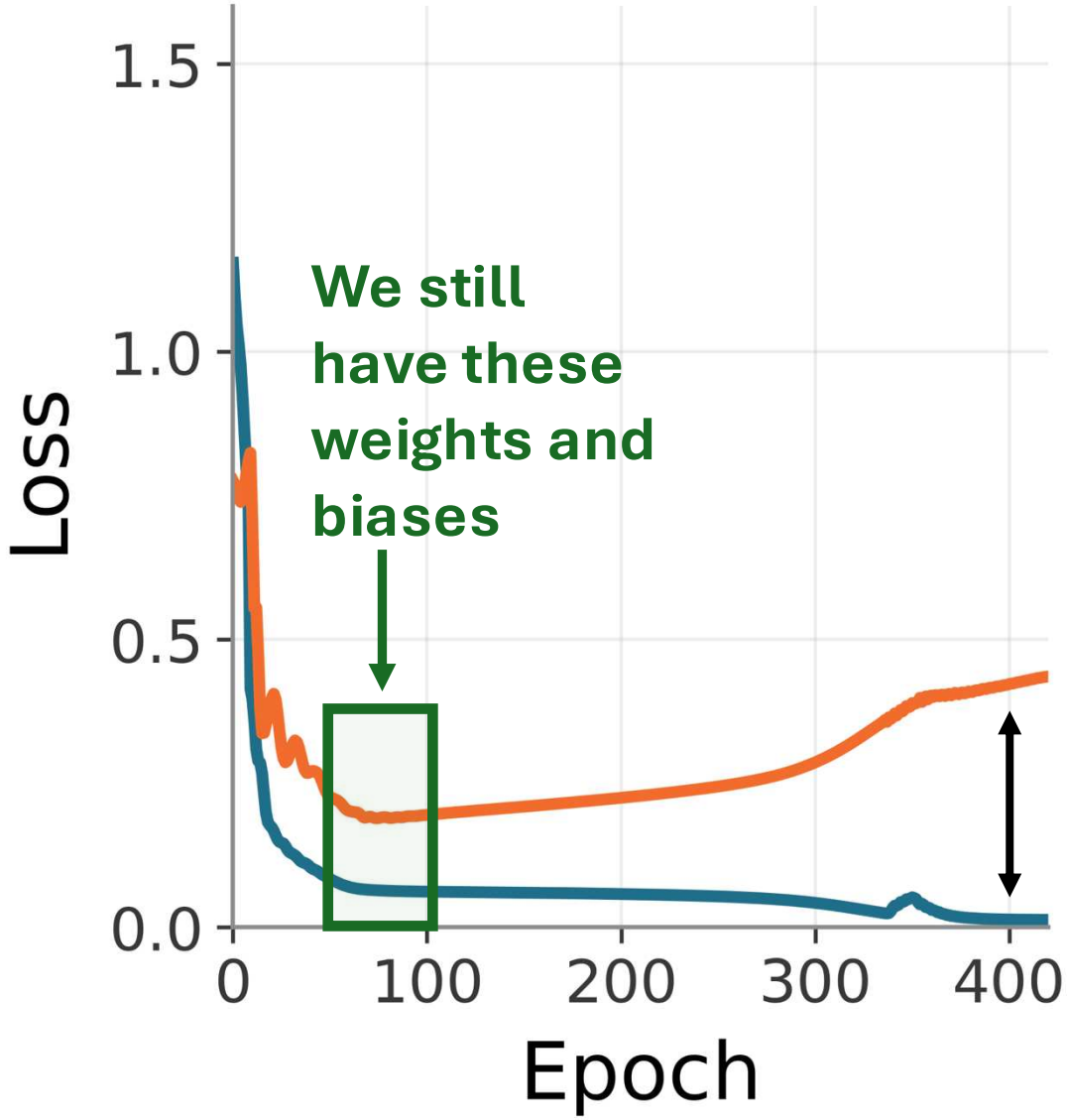


Remember **that 1 epoch = 1 complete pass** through the training set as we update weights using **backpropagation + gradient descent**





How to prevent overfitting?



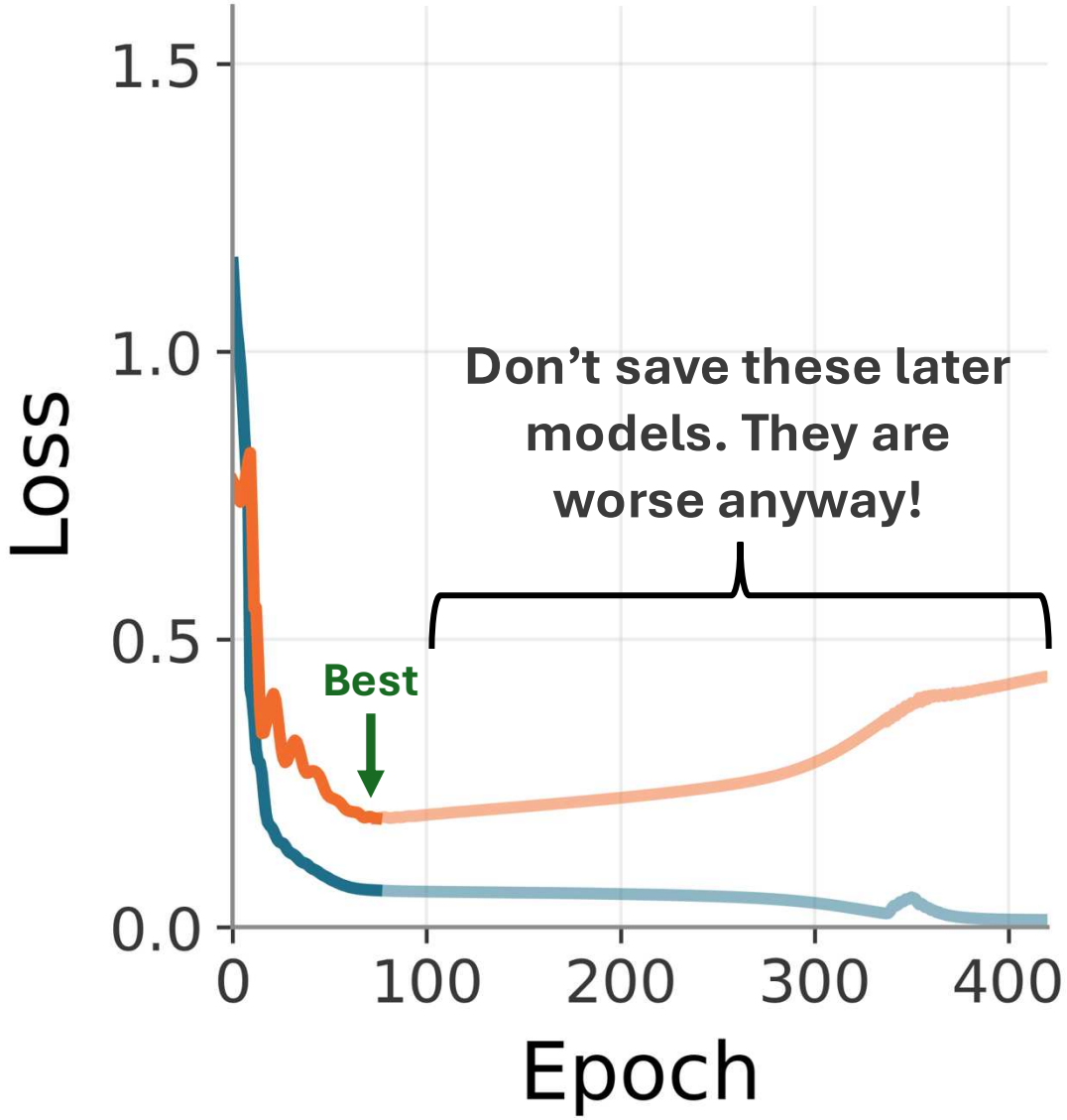
In this example, we trained our sample for about 400 epochs. But clearly...

The final model ≠ the best model

Why not just choose the model with the lowest validation error?

The **generalization gap** is the difference between the training and validation error. It gets larger as we overfit!

Early Stopping: Save the model *before* overfitting

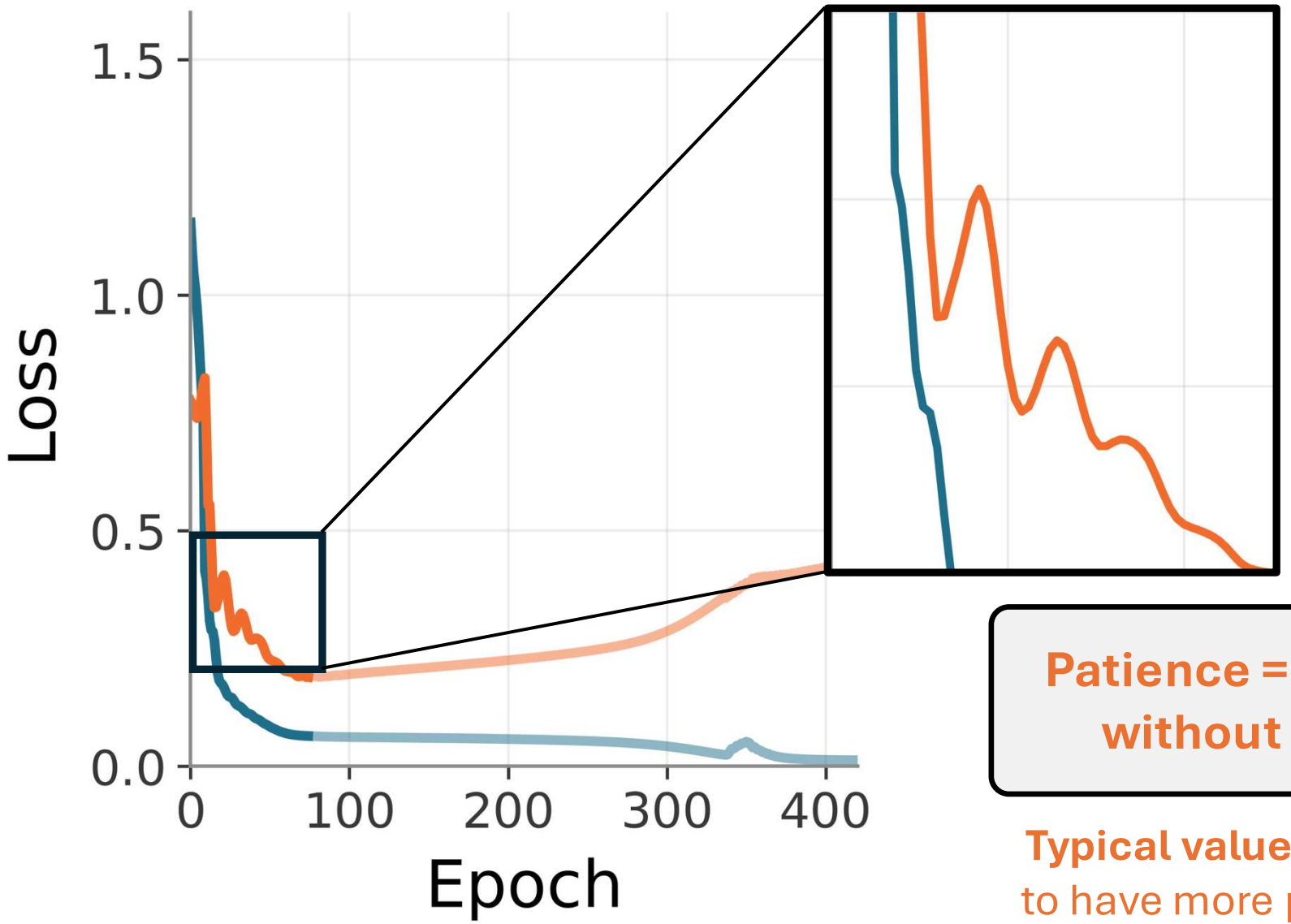


We don't know what these curves will look like before training through all the epochs, so the **early stopping procedure** is:

1. **Train the NN for an epoch**
2. **Compute the validation loss**
3. **If val loss improved, save the model**

If the validation loss has not improved for several epochs, stop training

The importance of being *patient*



Gradient descent is not perfect. Training and validation losses both often decrease "noisily"

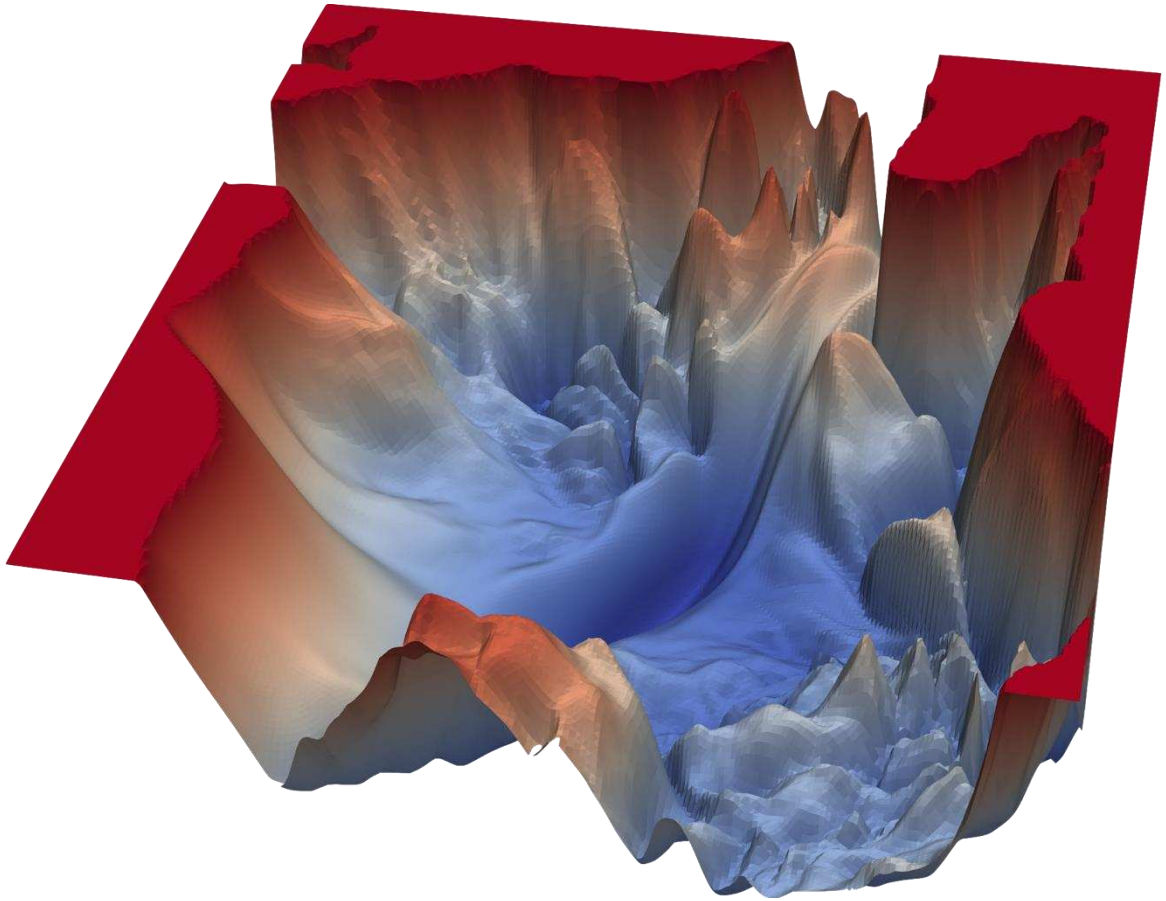
Just because val loss increases does not mean we should stop training!

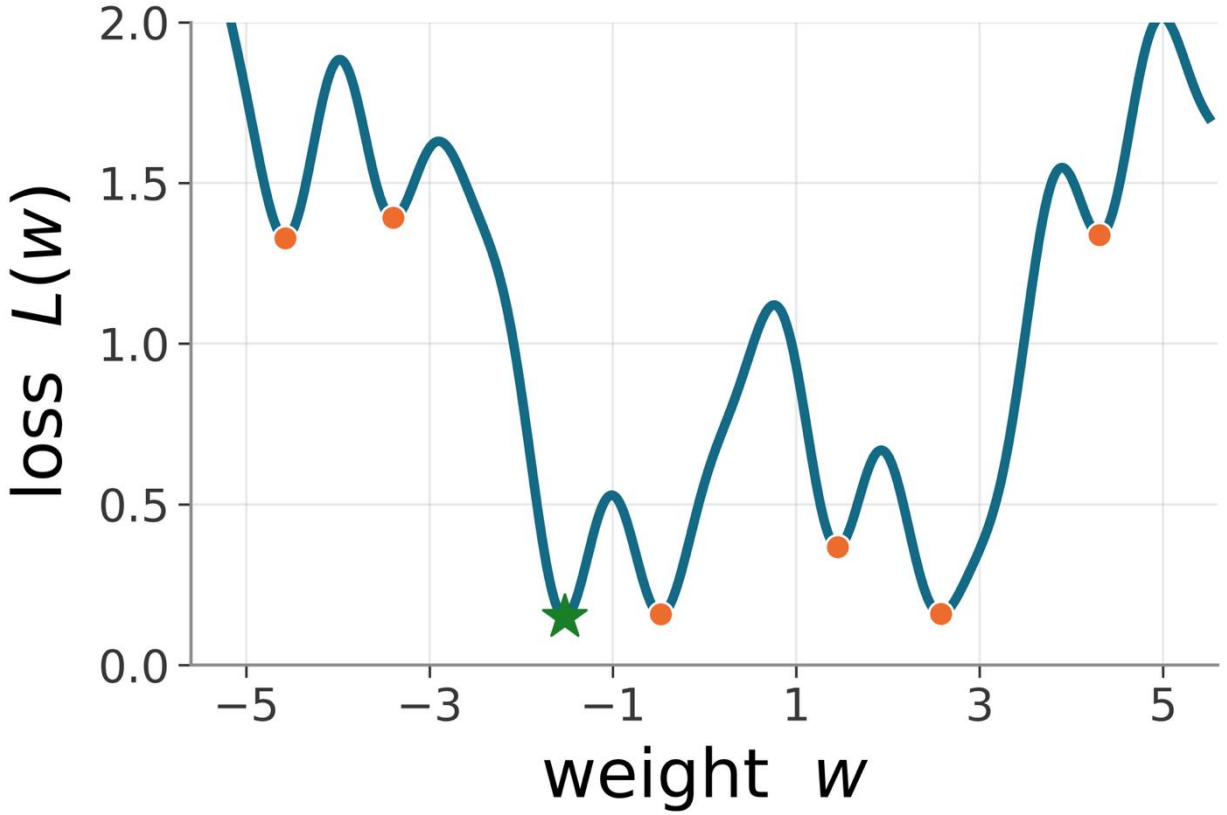
Patience = number of epochs allowed without validation improvement.

Typical values are 5-20 epochs...It *never hurts* to have more patience – it's just more expensive

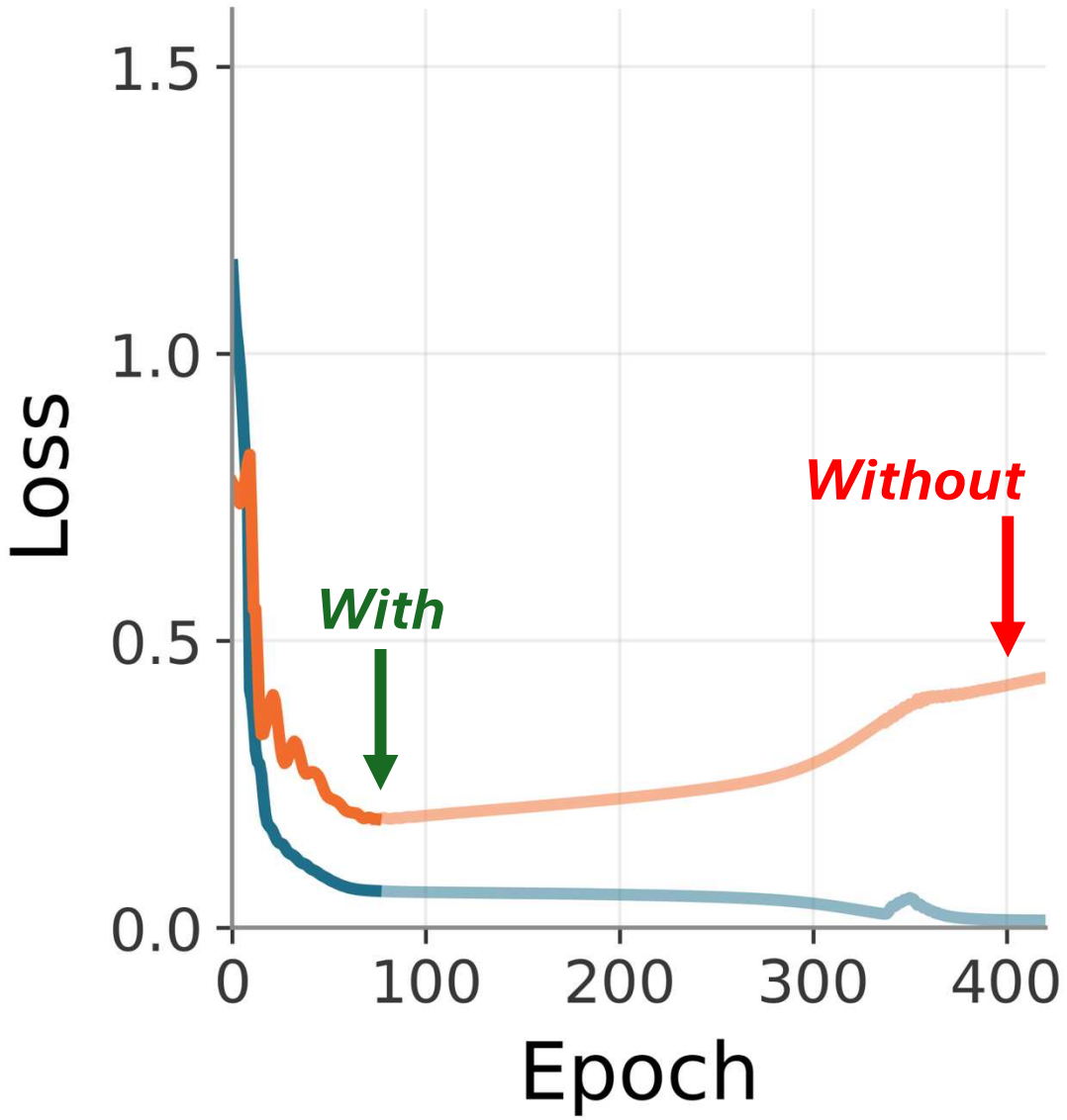
Give the NN time to escape local minima

Remember, NN loss landscapes are rough, and we might hop around local minima



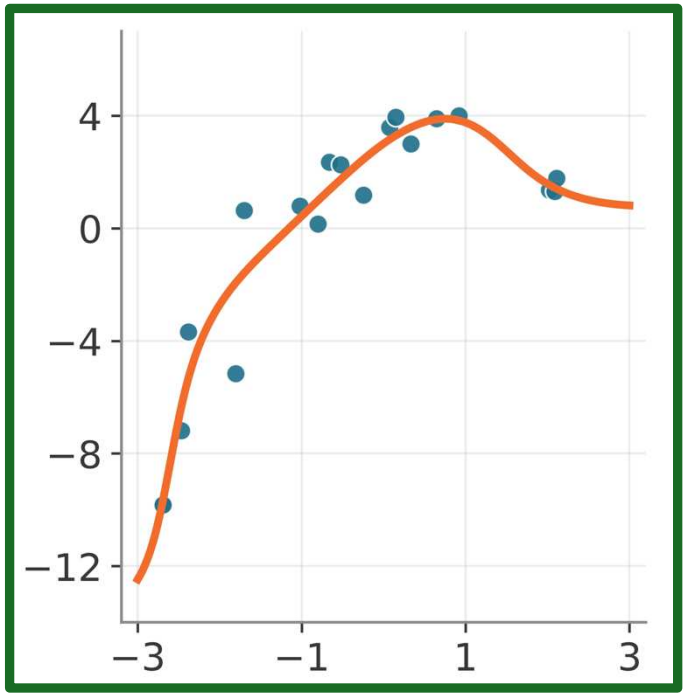


Today's topic is *regularization*

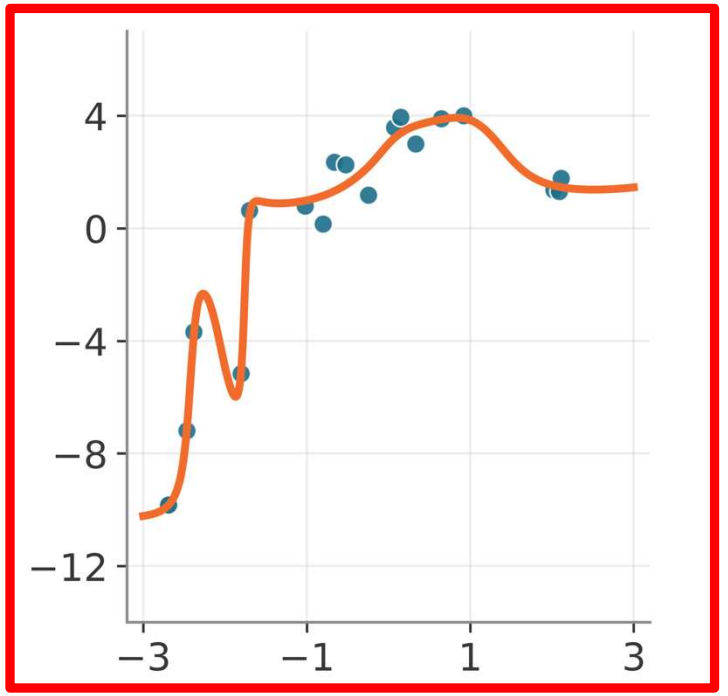


Is this really a form of regularization?

***With* early stopping**



***Without* early stopping**



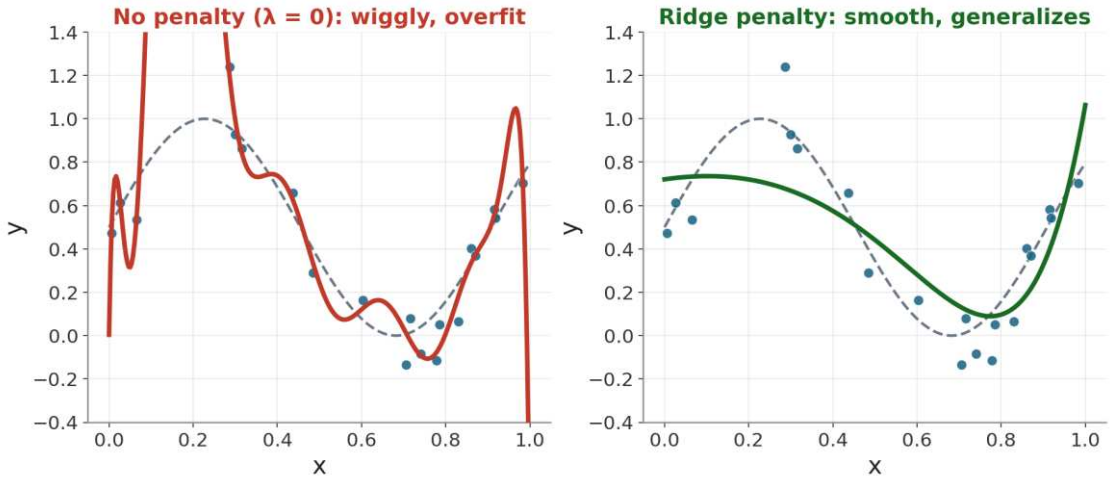
Yes – it helps keep our model simple!

How does this affect the weights (and biases)?

Recall: Past forms of regularization (non-NNs)

$$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_{16} x^{16}$$





In linear regression, we regularized by encouraging **small (near-zero) coefficients**



$$\mathcal{L} = \mathbf{MSE}(\beta_i) + \lambda R(\beta_i)$$

L_2-norm

$$R(\beta_i) = \sum_i \beta_i^2$$

Ridge regression

L_1-norm

$$R(\beta_i) = \sum_i |\beta_i|$$

Lasso regression

Early stopping indirectly keeps NN weights small

Weights are typically **initialized randomly** but with **small values near zero**. For example:

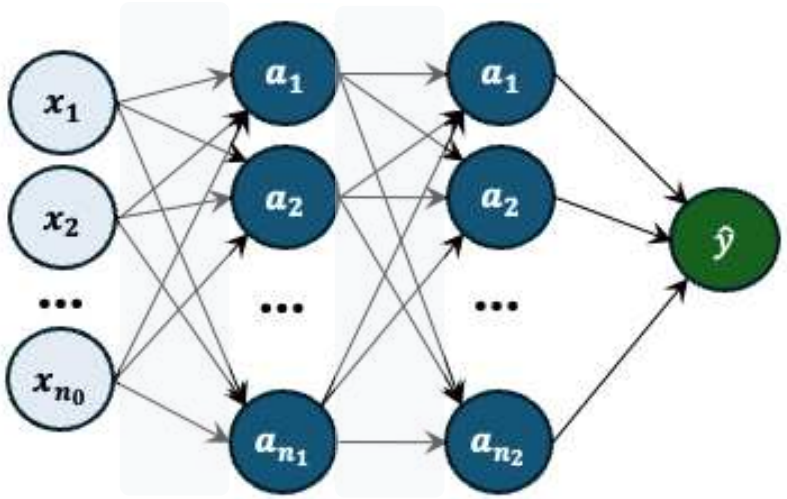
n_{in}: # input features

$$w_{ij} \sim \text{Normal}\left(0, \frac{2}{n_{\text{in}}}\right)$$

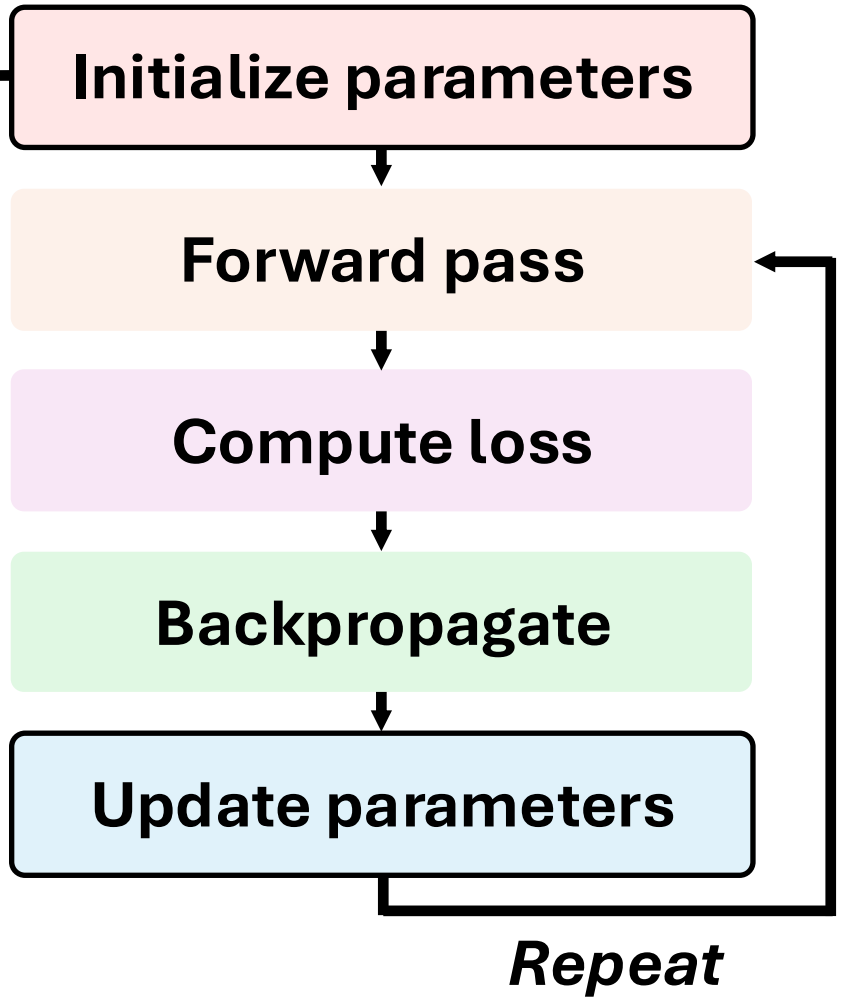
Sample weights from a Gaussian distribution (centered around zero)

With 100 input features, **weights are ~0.1**

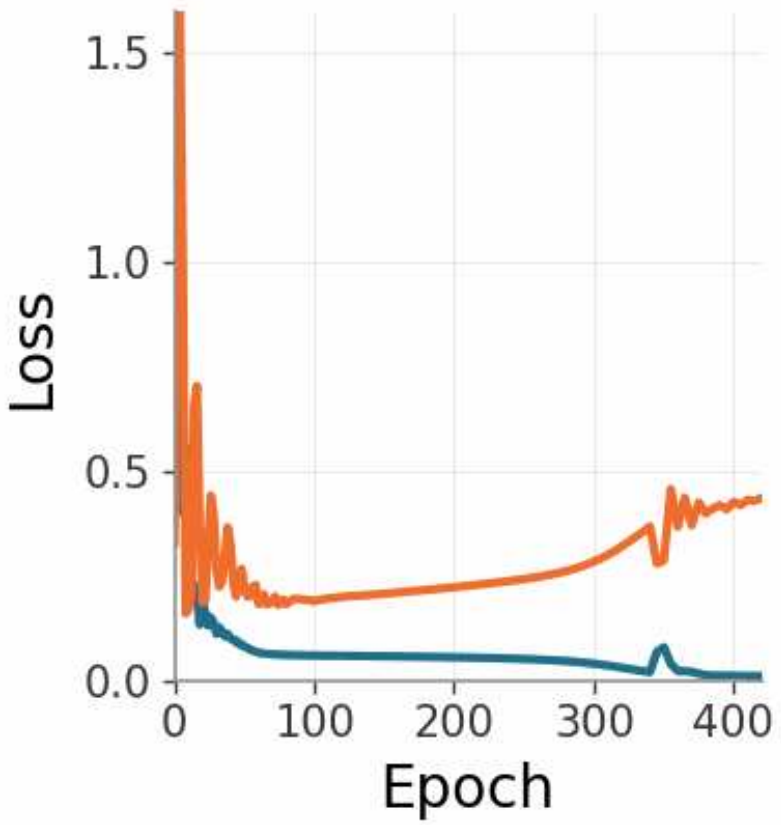
And biases usually start at zero

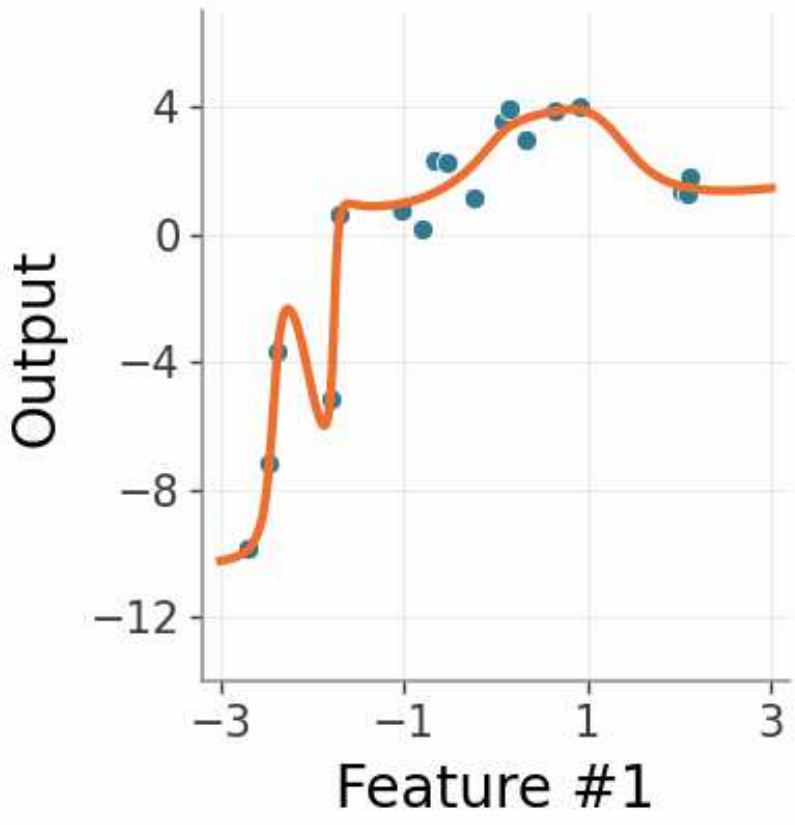


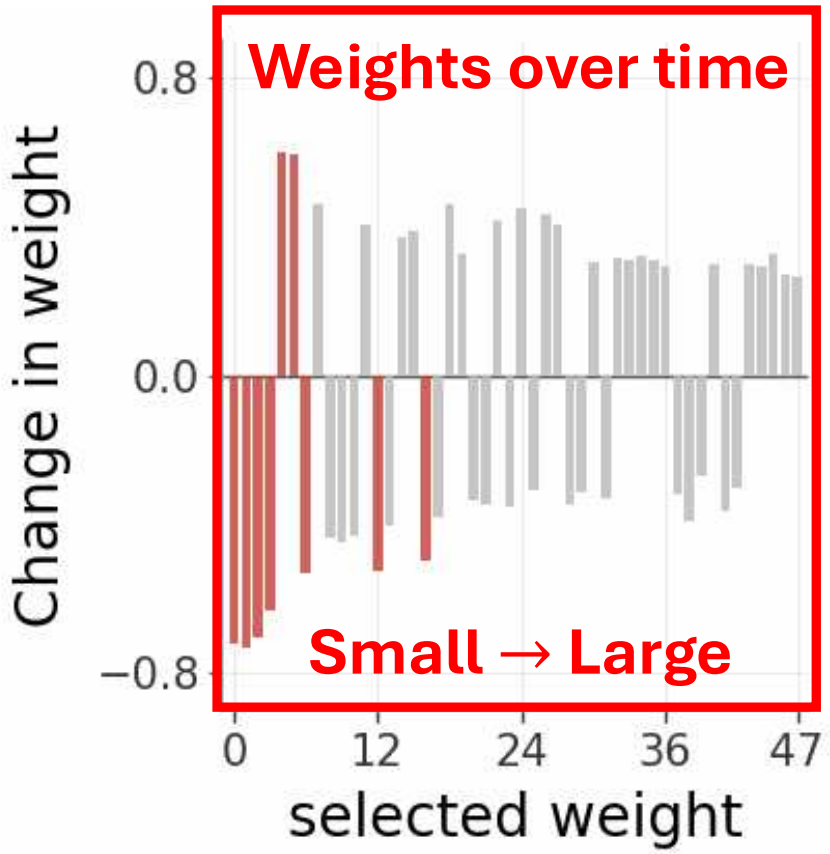
So in a way, we start off with a simple and regularized model, but **these weights grow as we train…**



Early stopping indirectly keeps NN weights small





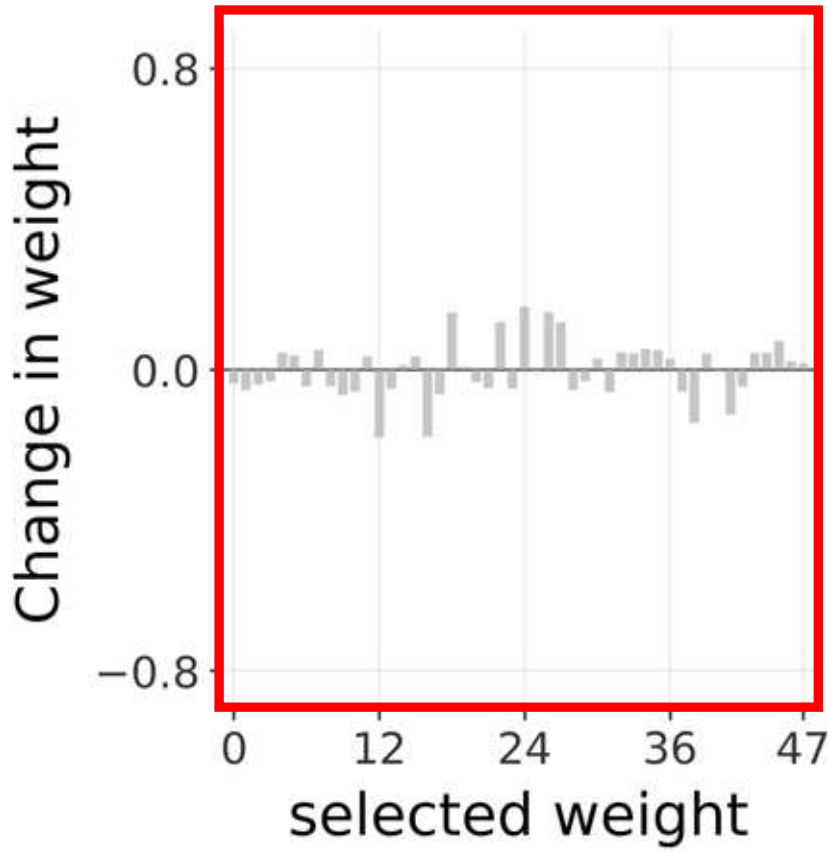


With early stopping, we "pause" the weights before they get too large

But early stopping alone is often insufficient

Limitations of early stopping:

- Some useful patterns are learned slowly. Stopping too soon **may prevent the model from fully learning complex patterns.**
- Early stopping **does not explicitly prevent weights from becoming large**, and with complex networks, weights can grow quickly!



→ **Early stopping is often used in tandem with other regularization techniques**

Questions?

L_1 and L_2 regularization for NNs

From linear regression to neural networks

Linear regression

$$\mathcal{L} = \mathbf{MSE}(\boldsymbol{\beta}_i) + \boldsymbol{\lambda} \boldsymbol{R}(\boldsymbol{\beta}_i)$$

L_2-norm

L_1-norm

$$\boldsymbol{R}(\boldsymbol{\beta}_i) = \sum_i \boldsymbol{\beta}_i^2$$

Ridge regression

$$\boldsymbol{R}(\boldsymbol{\beta}_i) = \sum_i |\boldsymbol{\beta}_i|$$

Lasso regression

Penalize large *coefficients*

From linear regression to neural networks

Linear regression

$$\mathcal{L} = \mathrm{MSE}(\beta_i) + \lambda R(\beta_i)$$

L_2-norm

$$R(\beta_i) = \sum_i \beta_i^2$$

Ridge regression

L_1-norm

$$R(\beta_i) = \sum_i |\beta_i|$$

Lasso regression

Neural networks

$$\mathcal{L} = \mathcal{L}_{\mathbf{data}}(w) + \lambda R(w)$$

L_2-norm

$$R(w_{ij}) = \sum_{i,j} w_{ij}^2$$

"Weight decay"

L_1-norm

$$R(w_{ij}) = \sum_{ij} |w_{ij}|$$

L1 penalty

Penalize large *coefficients* ⟷ Penalize large *weights*

From linear regression to neural networks

Let's start with **L2 penalty**, also known as **weight decay**, which is the probably the **most common regularization technique for neural networks** (in combination with early stopping)

Neural networks

$$\mathcal{L} = \mathcal{L}_{\text{data}}(w) + \lambda R(w)$$

L_2-norm

L_1-norm

$$R(w_{ij}) = \sum_{i,j} w_{ij}^2$$

"Weight decay"

$$R(w_{ij}) = \sum_{ij} |w_{ij}|$$

L1 penalty

L2 penalty: Weight decay

$$\mathcal{L} = \mathcal{L}_{\text{data}}(w_{ij}) + \lambda \sum_{i,j} w_{ij}^2$$

Penalize the squared sum of the NN weights*

Usual loss function: MSE for regression or cross-entropy for classification

This is the regularization strength. Larger λ = more penalty on complex models with large weights.

*We usually do *not* penalize large biases, just weights!

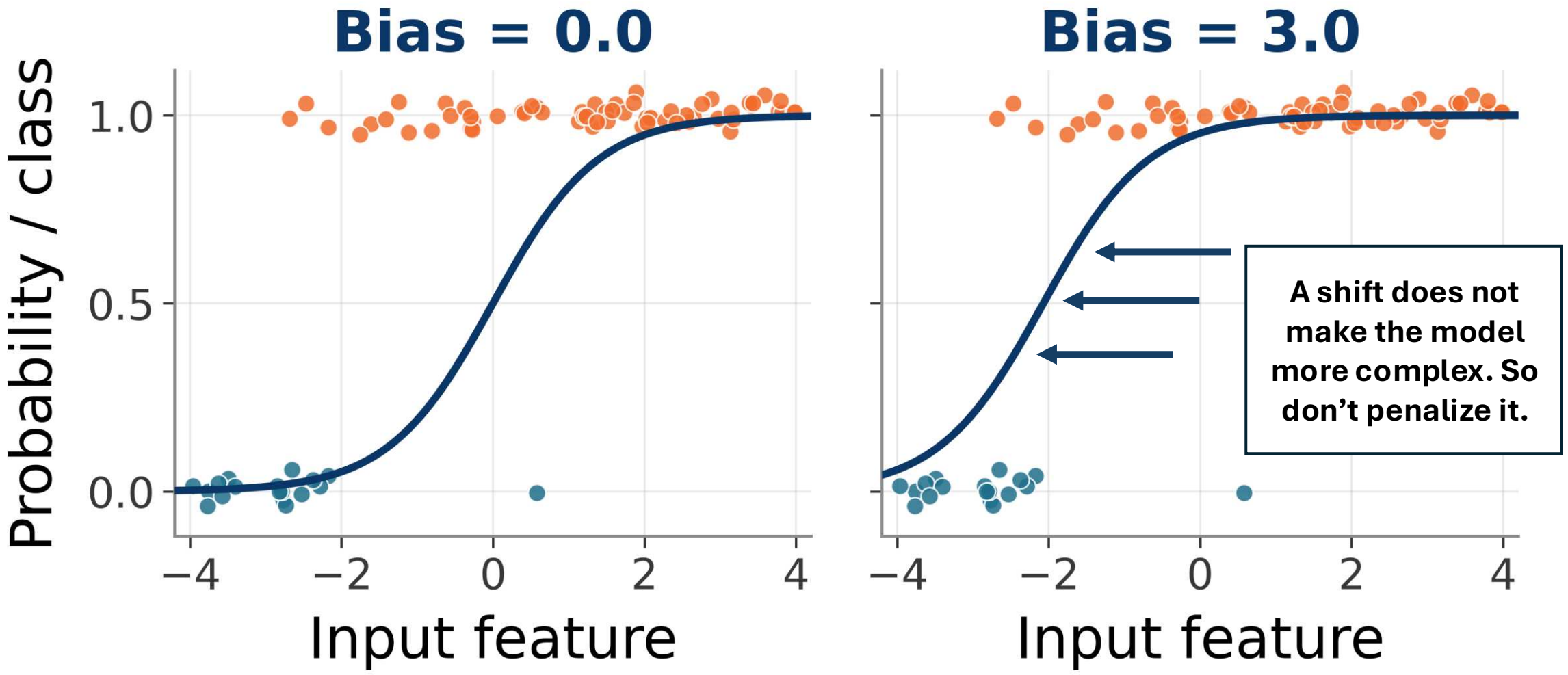
$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

Controls how sensitive model is to input

Controls the baseline offset

$$\begin{pmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \dots & w_{1n_{l-1}}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \dots & w_{2n_{l-1}}^{(l)} \\ \dots & \dots & \dots & \dots \\ w_{n_l 1}^{(l)} & w_{n_l 2}^{(l)} & \dots & w_{n_1 n_{l-1}}^{(l)} \end{pmatrix} \begin{pmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \dots \\ a_{n_{l-1}}^{(l)} \end{pmatrix} + \begin{pmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \dots \\ b_{n_l}^{(l)} \end{pmatrix}$$

Why penalize weights but *not* biases



Larger bias shifts the curve leftward to fit the data – we should *not* penalize this!

How adding an L2 penalty affects gradient descent

$$\mathcal{L} = \mathcal{L}_{\text{data}}(w_{ij}) + \lambda \sum_{i,j} w_{ij}^2$$

Remember:
w_{ij} is a weight feeding ***to i from j***

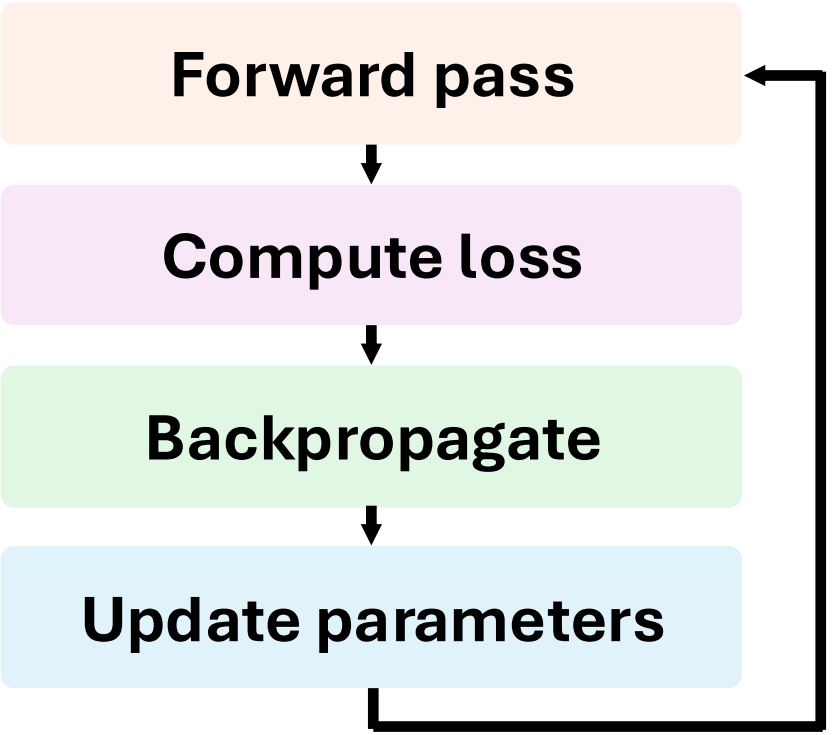
Gradient descent for each weight update:

$$w'_{ij} = w^0_{ij} - \eta \left(\frac{d\mathcal{L}}{dw_{ij}}\right)\Big|_{w_{ij}=w^0_{ij}}$$

$$\left(\frac{d\mathcal{L}}{dw_{ij}}\right)\Big|_{w_{ij}=w^0_{ij}} = \left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}}\right)\Big|_{w_{ij}=w^0_{ij}} + 2\lambda w^0_{ij}$$

This we get from backpropagation

Add a term for each weight *before* the update



How adding an L2 penalty affects gradient descent

$$w'_{ij} = w^0_{ij} - \eta \left[\left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}} \right) \Big|_{w_{ij}=w^0_{ij}} + 2\lambda w^0_{ij} \right]$$

Then let's distribute and group like terms…

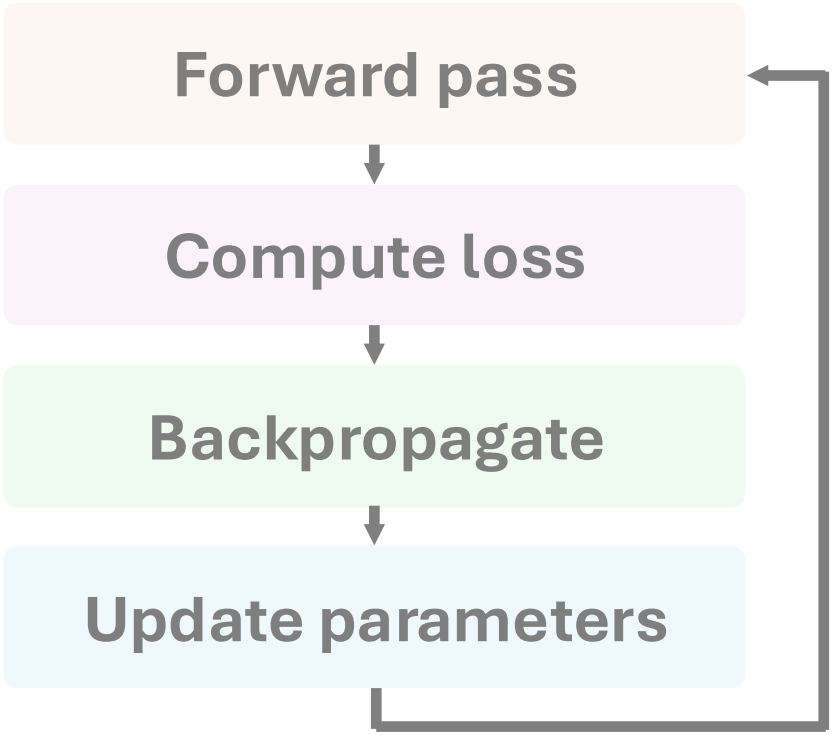
Gradient descent for each weight update:

$$w'_{ij} = w^0_{ij} - \eta \left(\frac{d\mathcal{L}}{dw_{ij}} \right) \Big|_{w_{ij}=w^0_{ij}}$$

$$\left(\frac{d\mathcal{L}}{dw_{ij}} \right) \Big|_{w_{ij}=w^0_{ij}} = \left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}} \right) \Big|_{w_{ij}=w^0_{ij}} + 2\lambda w^0_{ij}$$

This we get from backpropagation

Add a term for each weight *before* the update



How adding an L2 penalty affects gradient descent

$$w'_{ij} = w^0_{ij} - \eta \left[\left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}} \right) \Big|_{w_{ij}=w^0_{ij}} + 2\lambda w^0_{ij} \right] = (1 - 2\eta\lambda) w^0_{ij} - \eta \left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}} \right) \Big|_{w_{ij}=w^0_{ij}}$$

Gradient descent for each weight update:

$$w'_{ij} = w^0_{ij} - \eta \left(\frac{d\mathcal{L}}{dw_{ij}} \right) \Big|_{w_{ij}=w^0_{ij}}$$

$$\left(\frac{d\mathcal{L}}{dw_{ij}} \right) \Big|_{w_{ij}=w^0_{ij}} = \left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}} \right) \Big|_{w_{ij}=w^0_{ij}} + 2\lambda w^0_{ij}$$

This we get from backpropagation

Add a term for each weight *before* the update

- Forward pass
- Compute loss
- Backpropagate
- Update parameters

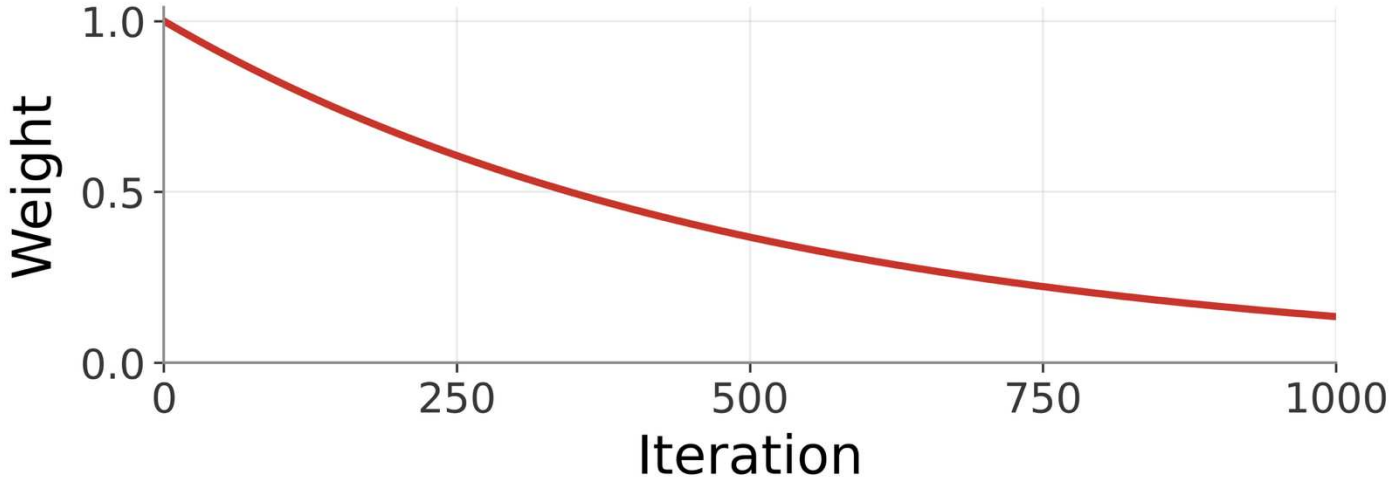
How adding an L2 penalty affects gradient descent

$$w'_{ij} = w^0_{ij} - \eta \left[\left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}} \right) \Big|_{w_{ij}=w^0_{ij}} + 2\lambda w^0_{ij} \right] = (1 - 2\eta\lambda) w^0_{ij} - \eta \left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}} \right) \Big|_{w_{ij}=w^0_{ij}}$$

We multiply the original weight by a **factor < 1**

For example, assume a small learning rate (η) of **0.01** and a regularization strength (λ) of **0.1**

$$(1 - 2\eta\lambda) = 0.998$$

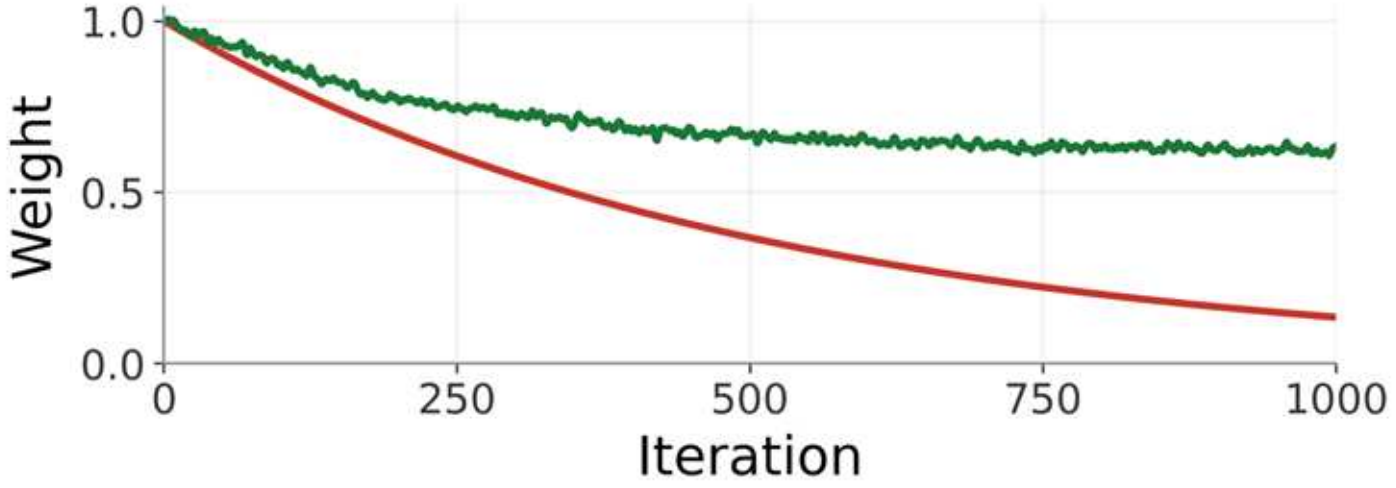


This means that all else being equal, the weight will get 0.2% smaller after each iteration → Hence the name ***weight decay*** for L2 penalty

How adding an L2 penalty affects gradient descent

$$w'_{ij} = w^0_{ij} - \eta\left[\left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}}\right)\Big|_{w_{ij}=w^0_{ij}} + 2\lambda w^0_{ij}\right] = (1-2\eta\lambda)w^0_{ij} - \eta\left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}}\right)\Big|_{w_{ij}=w^0_{ij}}$$

Of course, all else is usually *not* equal – since the weight affects the data loss



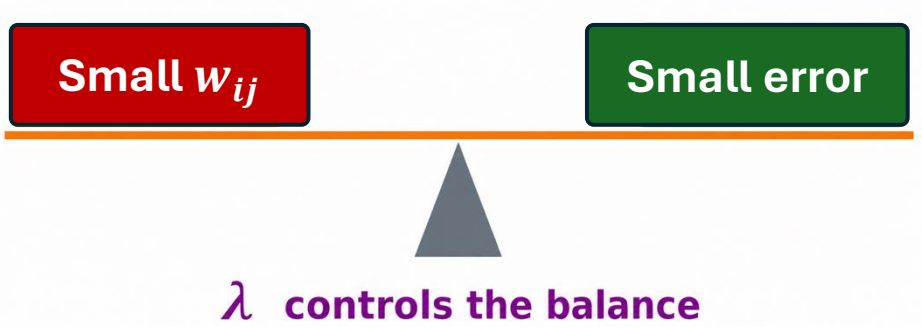
This means that all ***else being equal***, the weight will get 0.2% smaller after each iteration → Hence the name ***weight decay*** for L2 penalty

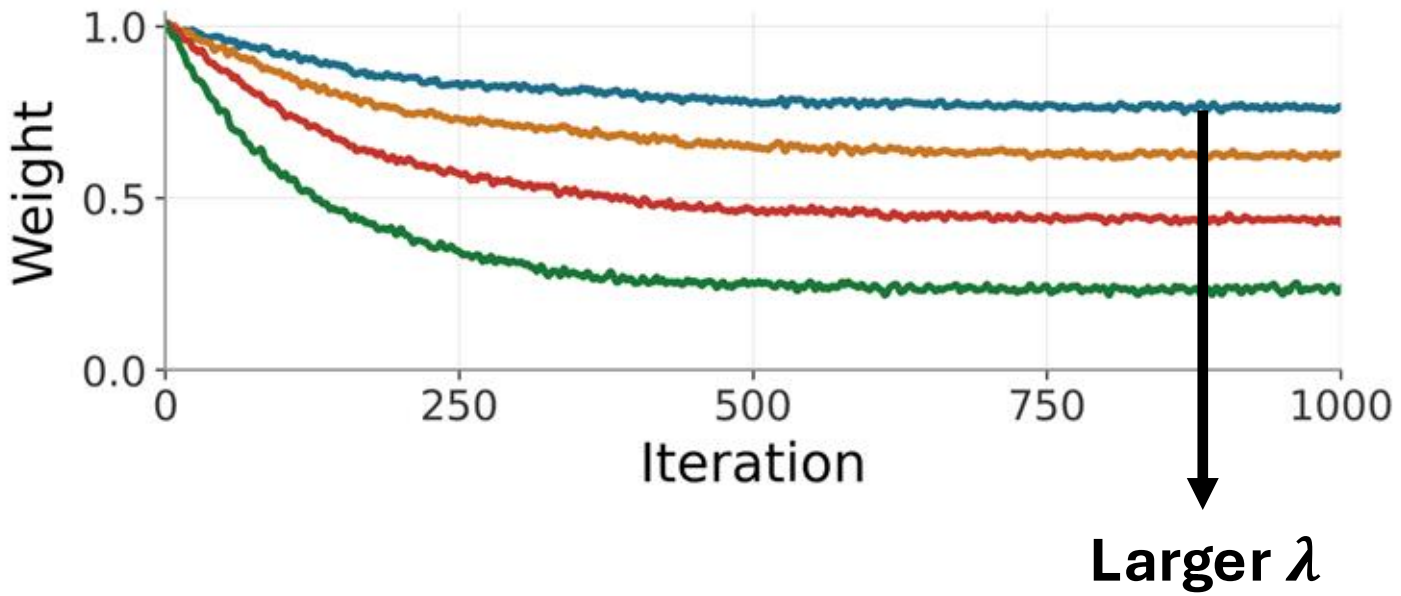
The regularization strength (λ)

There is a tradeoff between each weight's magnitude and its effect on the loss

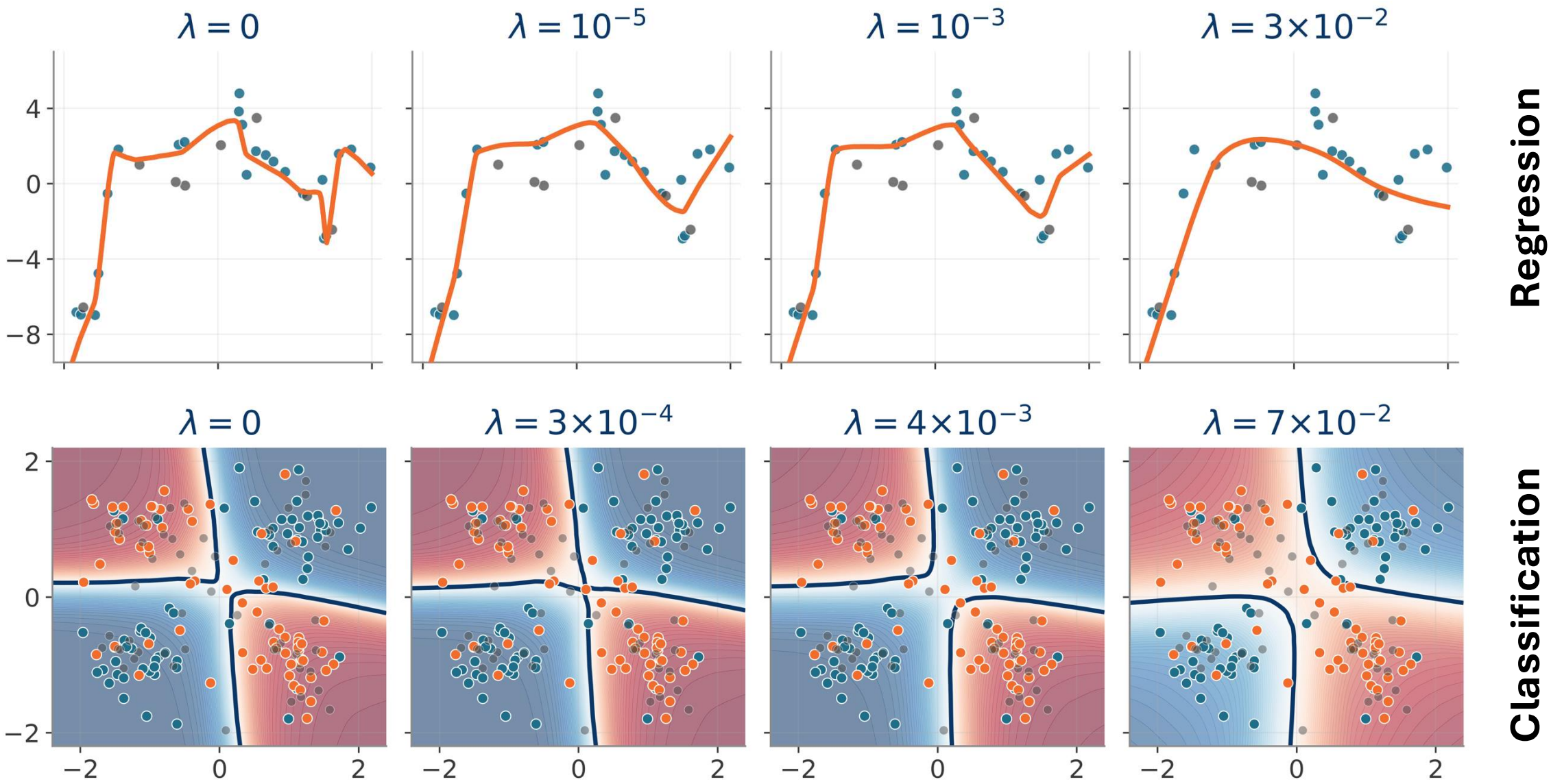
$$(1 - 2\eta\lambda)w_{ij}^0 - \eta\left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}}\right)\Bigg|_{w_{ij}=w_{ij}^0}$$

This tradeoff is controlled by the **regularization strength (λ),** a hyperparameter that *you* set





The regularization strength (λ)



Questions?

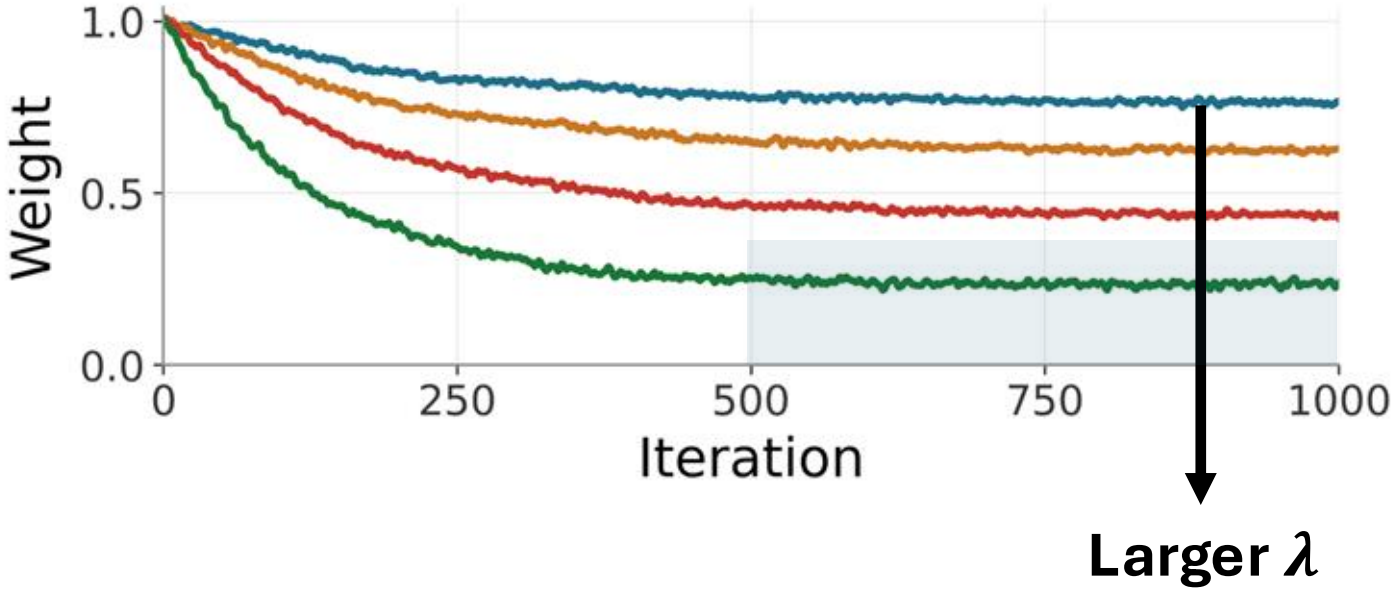
L2 regularization: Weights decay, but not to zero

If we ignore the data loss, each weight is some non-zero fraction of the previous weight. So it may *approach* zero but never reach it

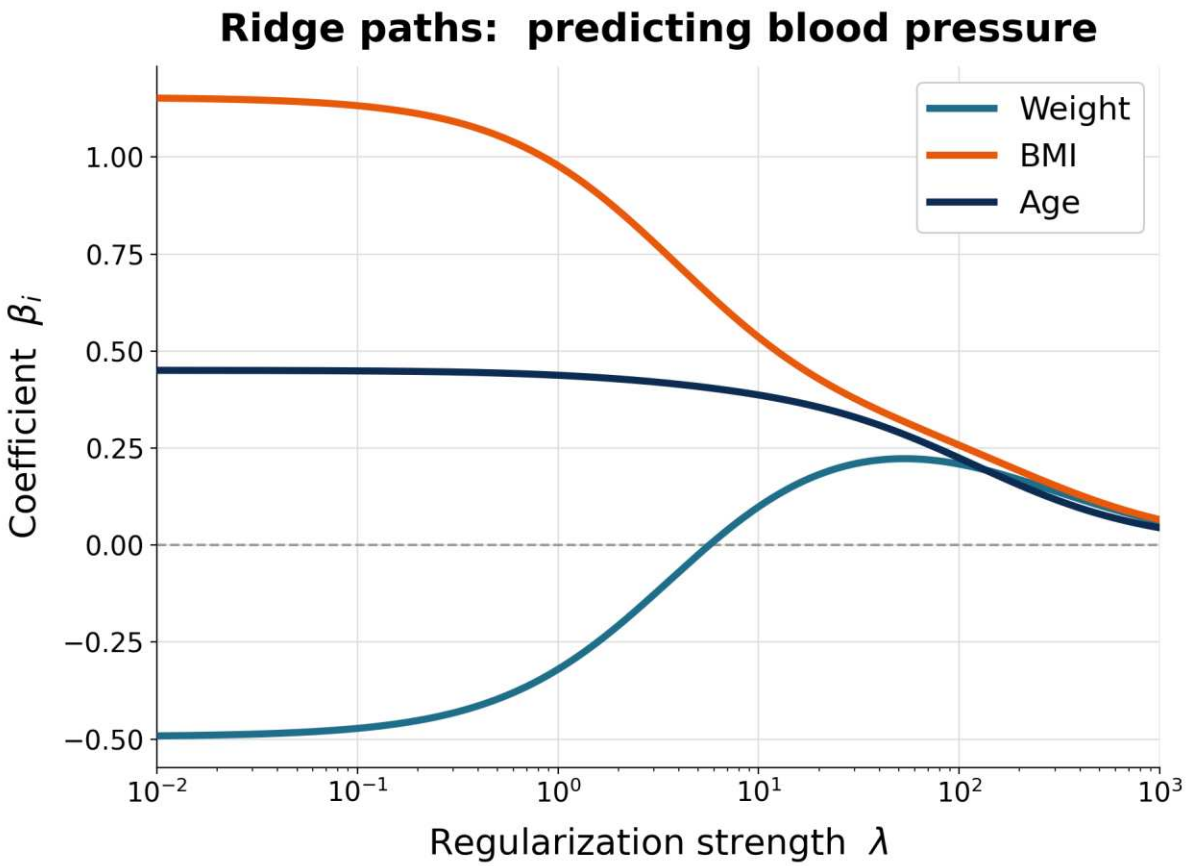
What if we want some weights to be completely "zeroed out"

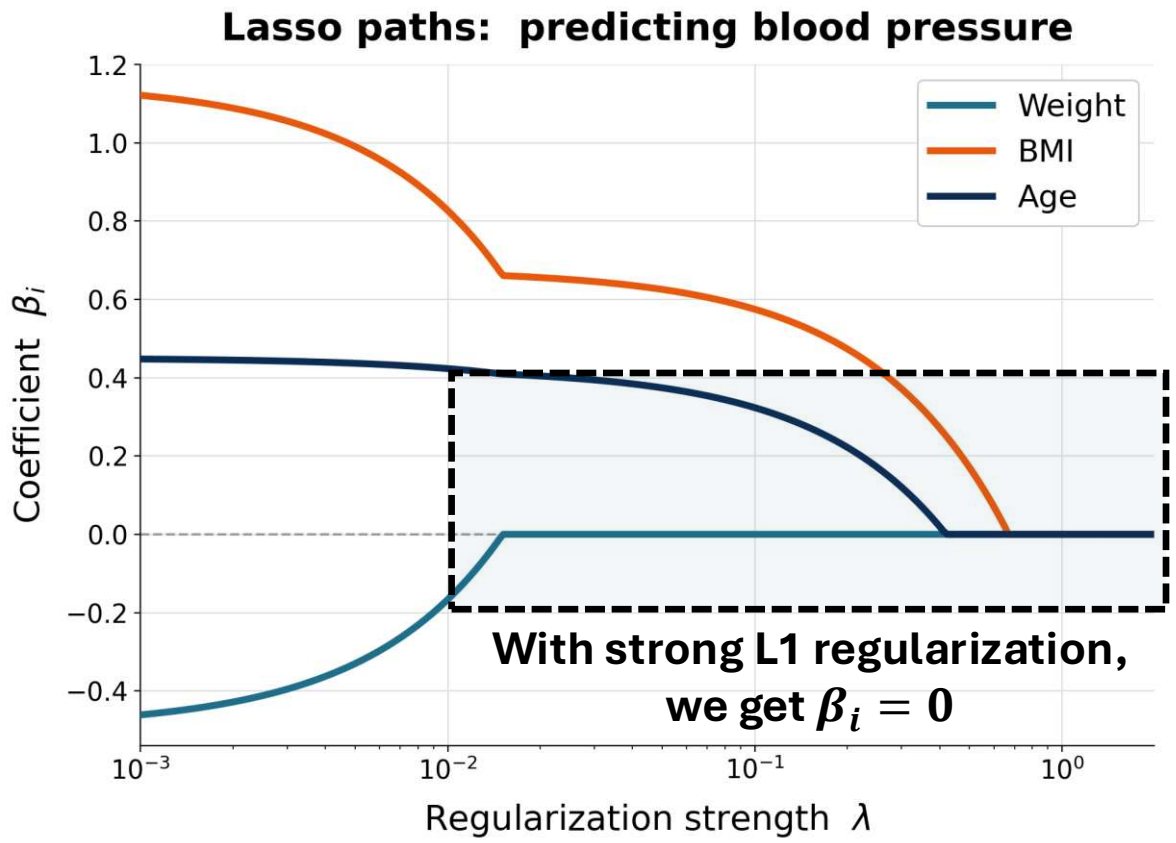
$$w_{ij} = (1 - 2\eta\lambda)w_{ij}^0 - \eta\left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}}\right)\Big|_{w_{ij}=w_{ij}^0}$$

Notice that even with strong regularization, the weight does not decay to zero – even if it does get very close…



Recall: β_i from Ridge (L2) *vs.* Lasso (L1)

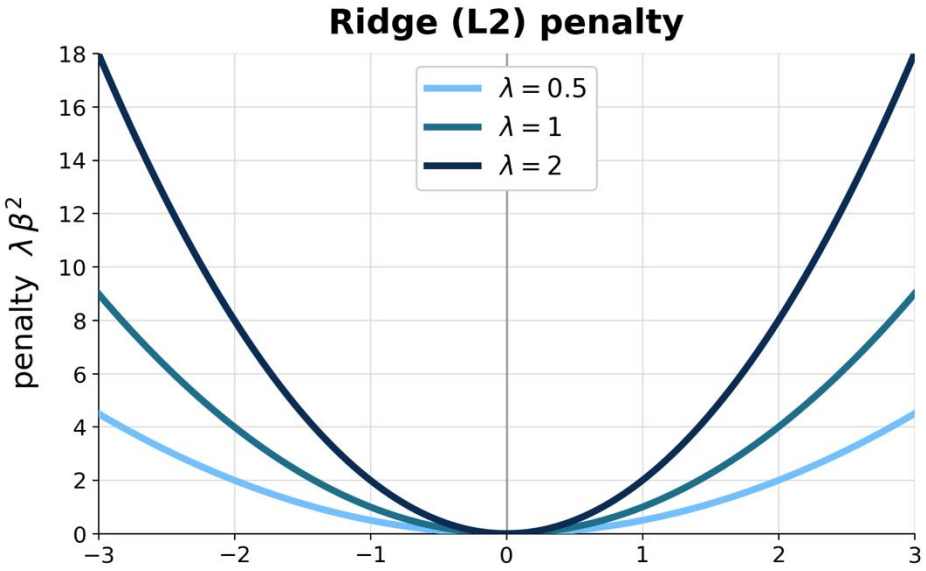


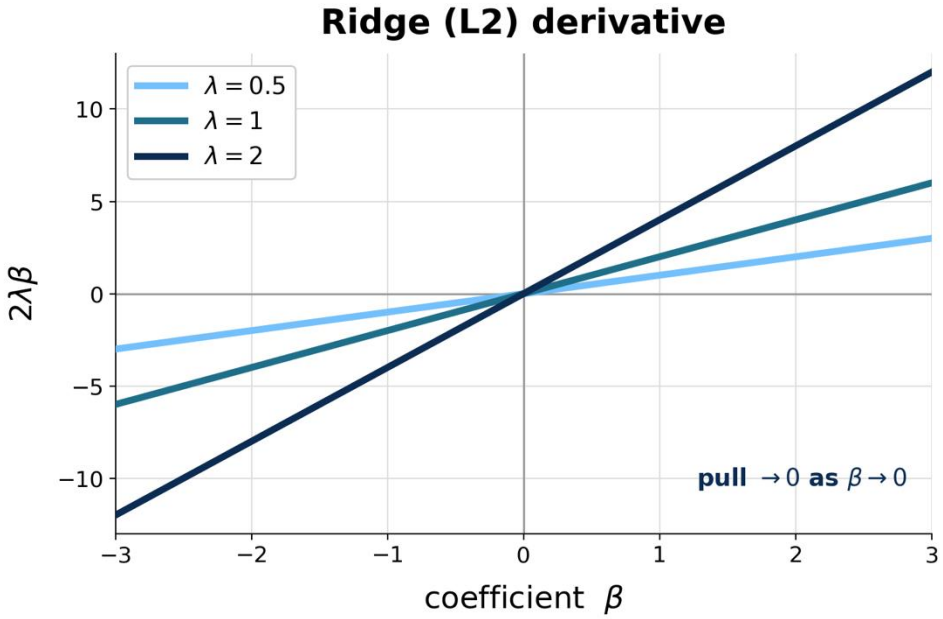


Ridge penalizes the largest coefficients, instead "spreading" out β_i while keeping *all* features

Lasso drops features one-by-one, setting $\beta_i = 0$ when features have little influence over data

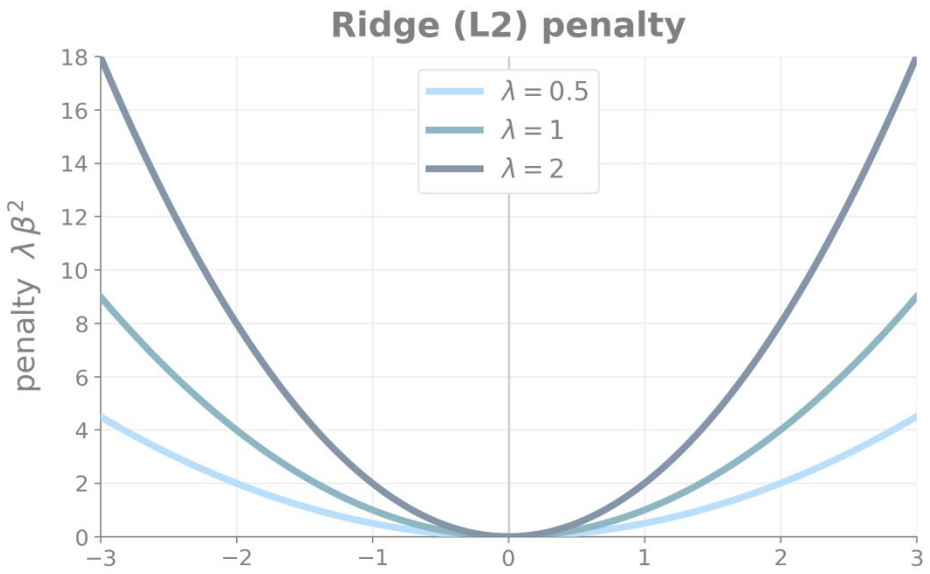
Recall: β_i from Ridge (L2) *vs.* Lasso (L1)

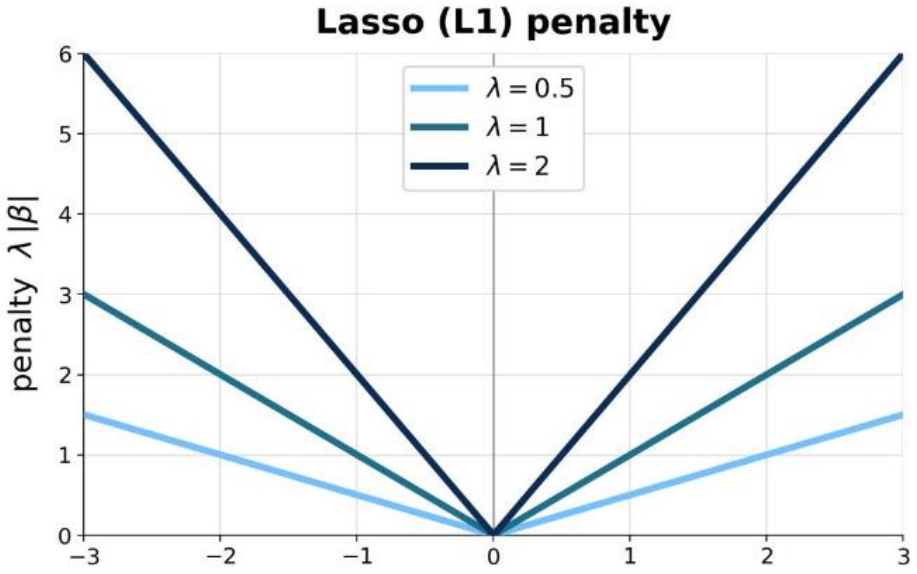


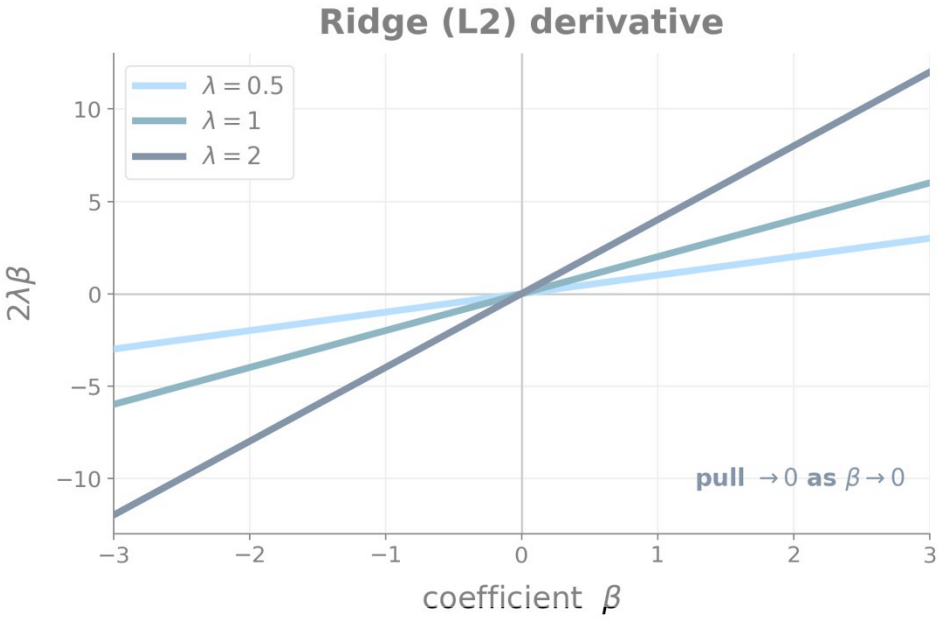


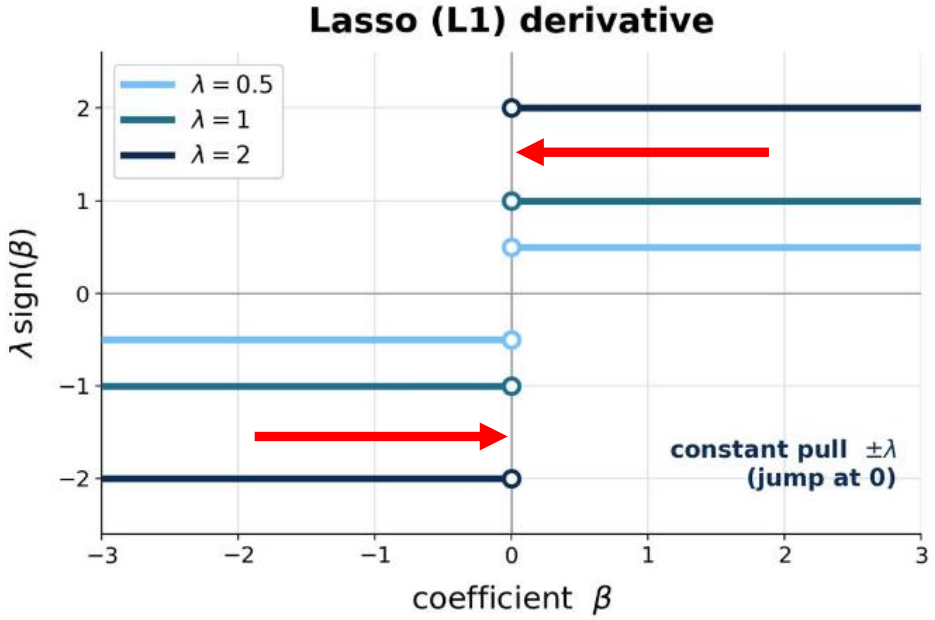
The **derivative**, and therefore the "force" during gradient descent, **goes to zero in L2 regularization**

Recall: β_i from Ridge (L2) *vs.* Lasso (L1)









The force remains constant in **L1 regularization**
→ there is a **persistent force to shrink coefficients**

The same idea holds for NNs: L1 penalty

$$\mathcal{L} = \mathcal{L}_{\text{data}}(w_{ij}) + \lambda \sum_{i,j} |w_{ij}|$$

← Penalize the sum of the absolute weight values

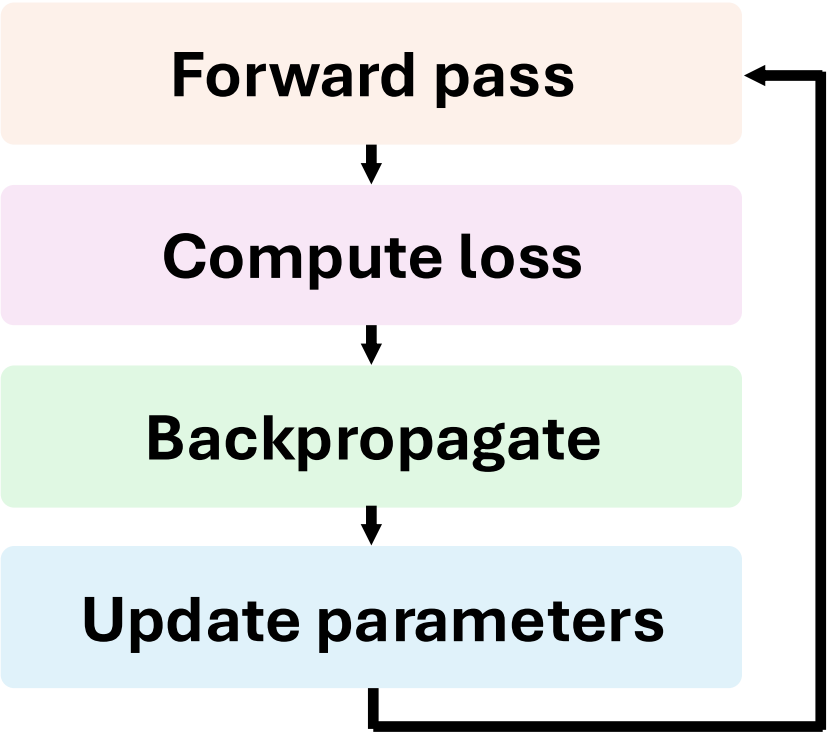
Gradient descent for each weight update:

$$w'_{ij} = w^0_{ij} - \eta \left(\frac{d\mathcal{L}}{dw_{ij}} \right) \Bigg|_{w_{ij}=w^0_{ij}}$$

$$\left(\frac{d\mathcal{L}}{dw_{ij}} \right) \Bigg|_{w_{ij}=w^0_{ij}} = \left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}} \right) \Bigg|_{w_{ij}=w^0_{ij}} \boxed{\pm \lambda}$$

This we get from backpropagation

Add ***or*** subtract a **fixed** term for each weight



How adding L1 penalty affects gradient descent

$$w'_{ij} = w^0_{ij} - \eta \left[\left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}} \right) \Bigg|_{w_{ij}=w^0_{ij}} \pm \lambda \right]$$

Gradient descent for each weight update:

$$w'_{ij} = w^0_{ij} - \eta \left(\frac{d\mathcal{L}}{dw_{ij}} \right) \Bigg|_{w_{ij}=w^0_{ij}}$$

$$\left(\frac{d\mathcal{L}}{dw_{ij}} \right) \Bigg|_{w_{ij}=w^0_{ij}} = \left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}} \right) \Bigg|_{w_{ij}=w^0_{ij}} \pm \lambda$$

This we get from backpropagation

Add or subtract a **fixed** term for each weight

How adding L1 penalty affects gradient descent

$$w'_{ij} = w^0_{ij} - \eta \left[\left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}} \right)\Big|_{w_{ij}=w^0_{ij}} \pm \lambda \right] = w^0_{ij} - \eta \left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}} \right)\Big|_{w_{ij}=w^0_{ij}} \mp \eta\lambda$$

The magnitude of this term does not depend on the weight itself

But its sign does.

Positive if $w^0_{ij} < 0$ and negative if $w^0_{ij} > 0$

Push toward zero

Gradient descent for each weight update:

$$w'_{ij} = w^0_{ij} - \eta \left(\frac{d\mathcal{L}}{dw_{ij}} \right)\Big|_{w_{ij}=w^0_{ij}}$$

$$\left(\frac{d\mathcal{L}}{dw_{ij}} \right)\Big|_{w_{ij}=w^0_{ij}} = \left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}} \right)\Big|_{w_{ij}=w^0_{ij}} \pm \lambda$$

This we get from backpropagation

Add or subtract a **fixed** term for each weight

How adding L1 penalty affects gradient descent

$$w'_{ij} = w^0_{ij} - \eta\left[\left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}}\right)\Big|_{w_{ij}=w^0_{ij}} \pm \lambda\right] = w^0_{ij} - \eta\left(\frac{d\mathcal{L}_{\text{data}}}{dw_{ij}}\right)\Big|_{w_{ij}=w^0_{ij}} \mp \eta\lambda$$

Suppose this = 0

For example, consider a weight $w_{ij} = -0.25$ whose value does not affect the data loss.

$$w'_{ij} = w^0_{ij} + \eta\lambda$$

Negative weight, add to it

Then, with a learning rate (η) of **0.01** and a regularization strength (λ) of **0.1**, we get:

$$w'_{ij} = -0.25 + (0.01)(0.1) = -0.249$$

Weight tends toward zero

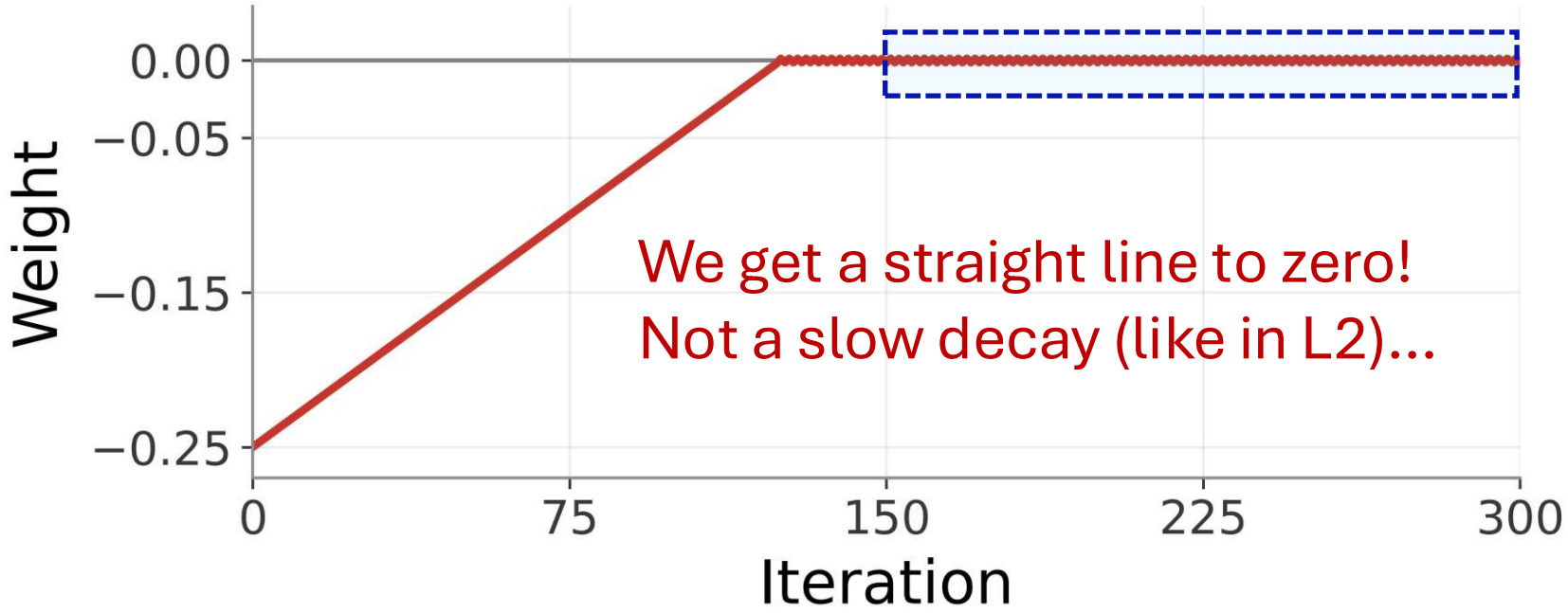
The magnitude of this term does not depend on the weight itself

But its sign does.

Positive if $w^0_{ij} < 0$ and negative if $w^0_{ij} > 0$

Push toward zero

How adding L1 penalty affects gradient descent



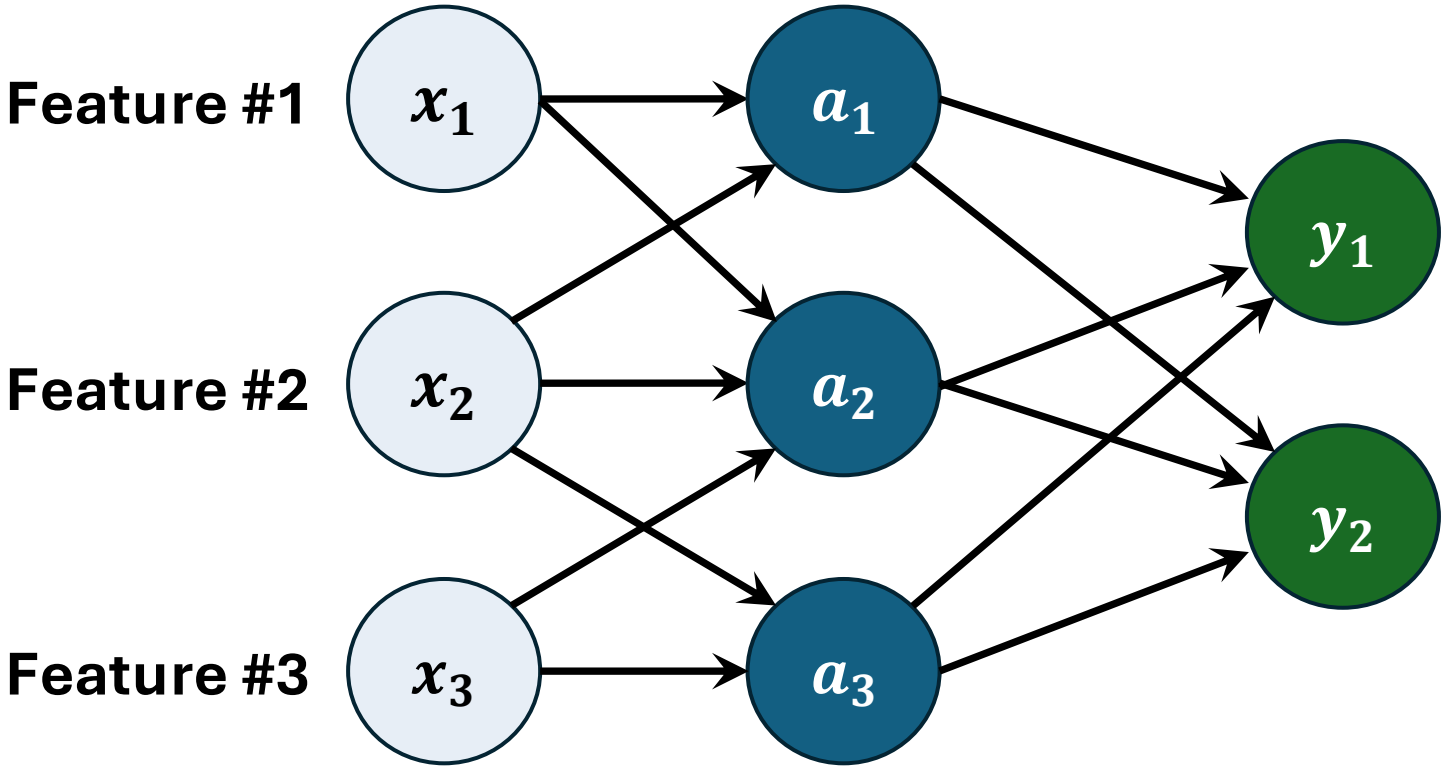
Then, with a learning rate (η) of **0.01** and a regularization strength (λ) of **0.1**, we get:

$$w'_{ij} = -0.25 + 2(0.01)(0.1) = -0.248$$

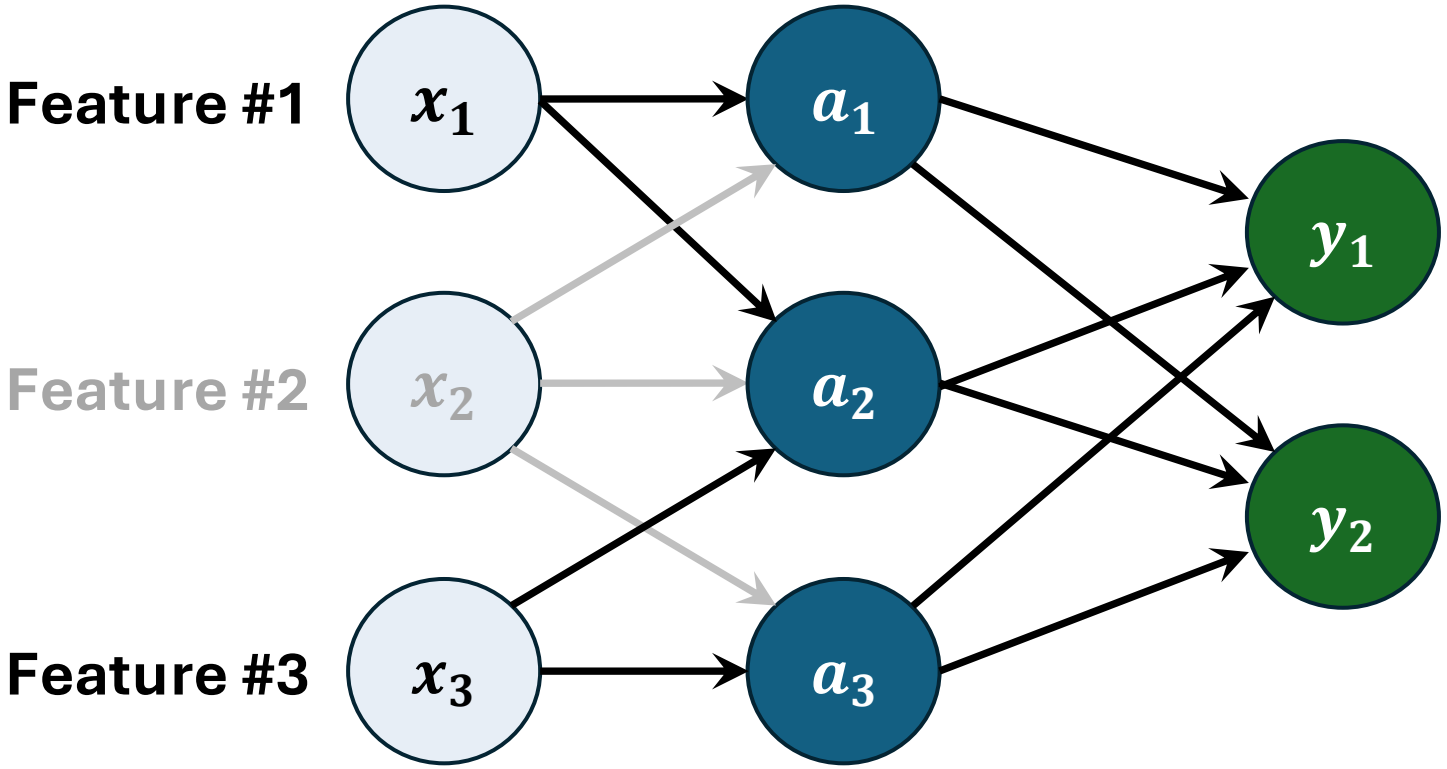
Weight tends toward zero

Weights are usually not *exactly* zero but have small fluctuations. We can round them to zero if we'd like to drop weights entirely.

L1 regularization creates *sparse* neural networks



L1 regularization creates *sparse* neural networks



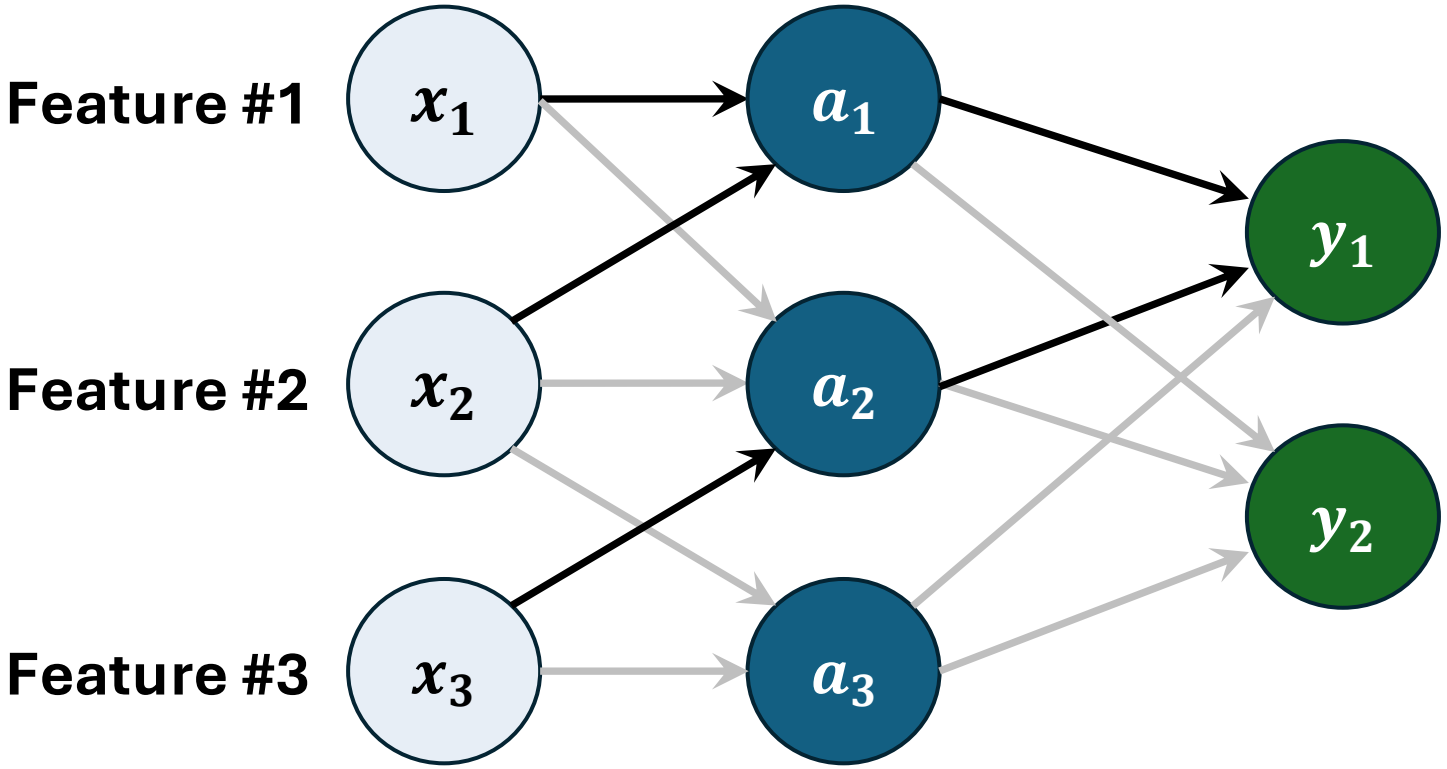
Suppose with L1 penalty, we find that:

$$w_{12}^{(1)} = w_{22}^{(1)} = w_{32}^{(1)} = 0$$

Then **Feature #2** is not an important predictor for our outputs – **drop it** entirely!

L1 regularization can be useful for *feature selection*

L1 regularization creates *sparse* neural networks



With all that said: if all you care about is model accuracy, then L2 regularization is usually the better choice…**Permanently removing entire connections makes the model less flexible.**

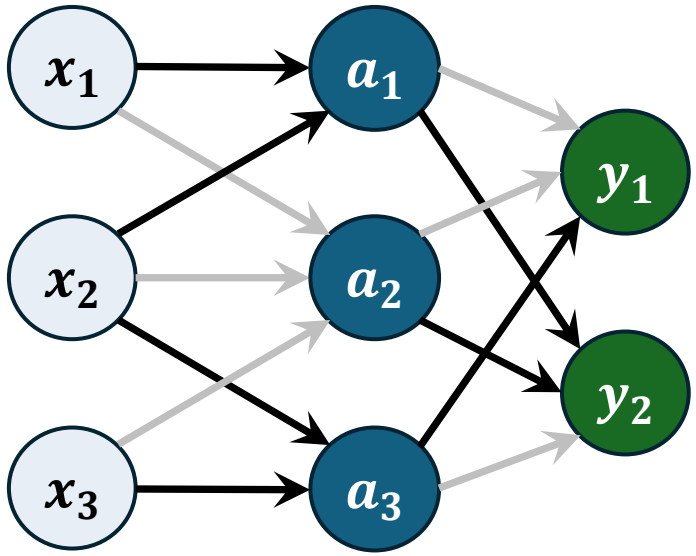
Even if we don't remove any features, it can be useful to have **fewer connections** between layers:

- More interpretable (*e.g.*, x_3 affects y_1)
- Less model storage
- Potentially faster inference (sparse matrix multiplication)

Dropout

From *permanent* sparsity to ***temporary*** **removal of connections**

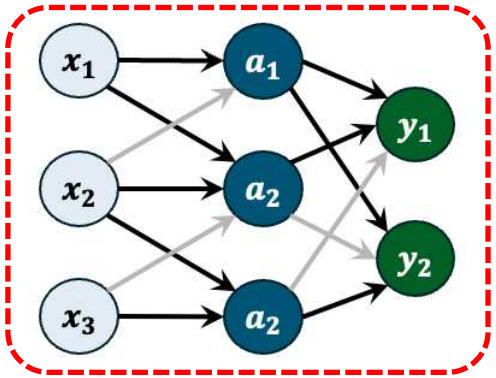
Dropout *varies* which connections are removed



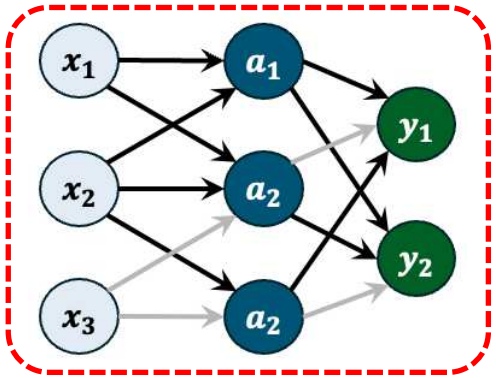
With L1 regularization, training leads to $w_{ij} \approx 0$ for certain weights.

These "zeroed-out" connections don't contribute at inference.

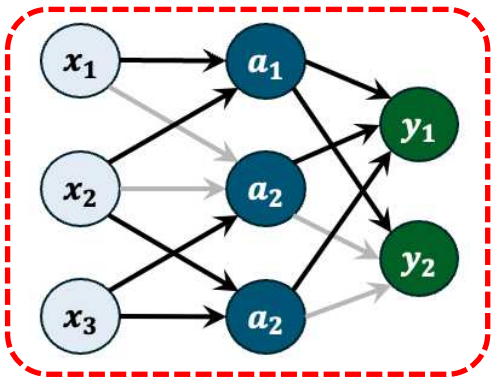
Forward Pass #1



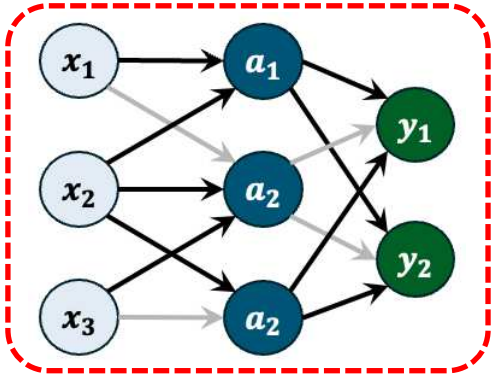
Forward Pass #2



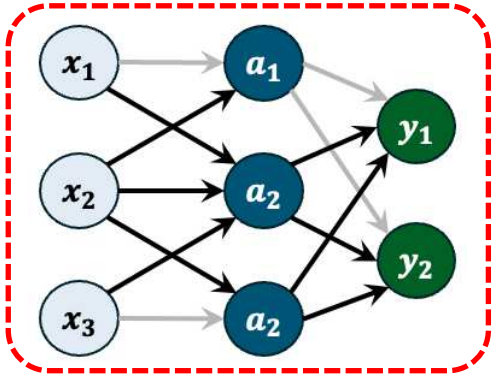
Forward Pass #3



Forward Pass #4



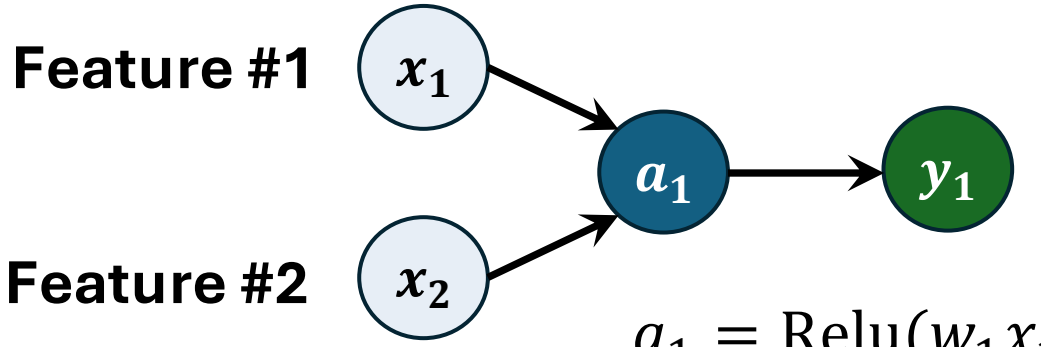
Forward Pass #5



With **dropout**, we **randomly exclude** some connections ***during training***

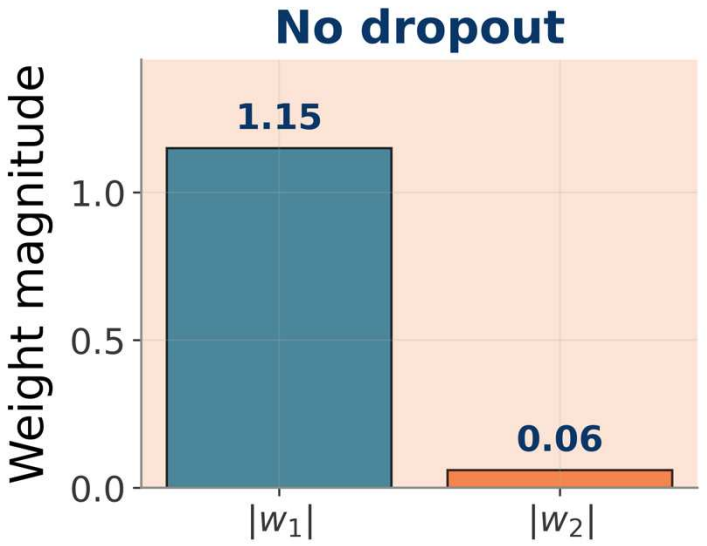
But then after training, ***at inference***, we use **all connections** to make a prediction

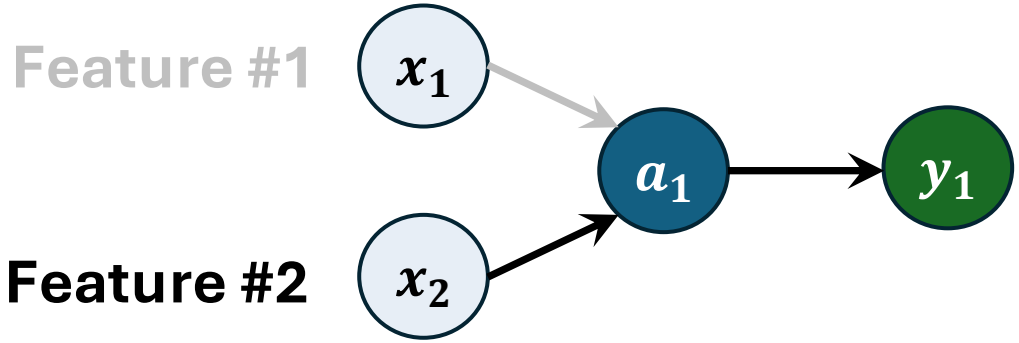
Why dropout helps, conceptually

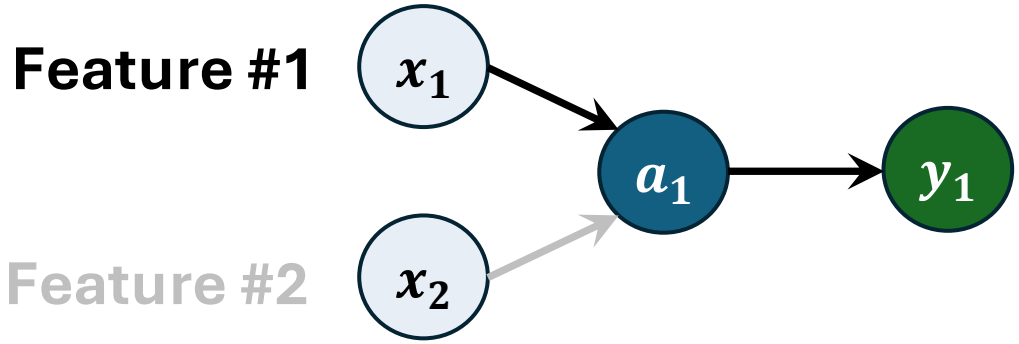


$$a_1 = \text{Relu}(w_1 x_1 + w_2 x_2 + b)$$

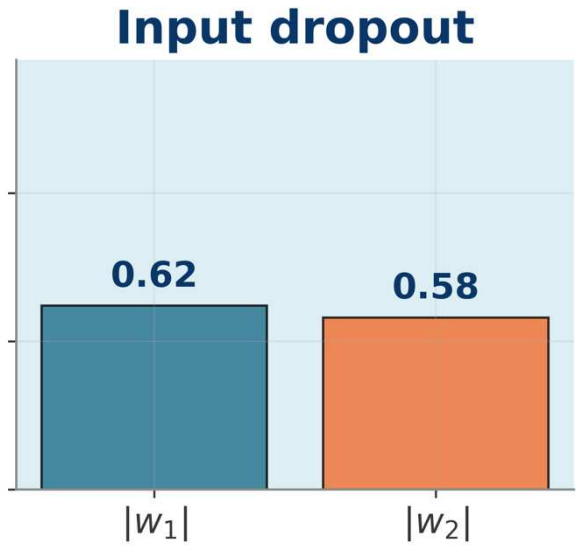
Heavy reliance on feature #1



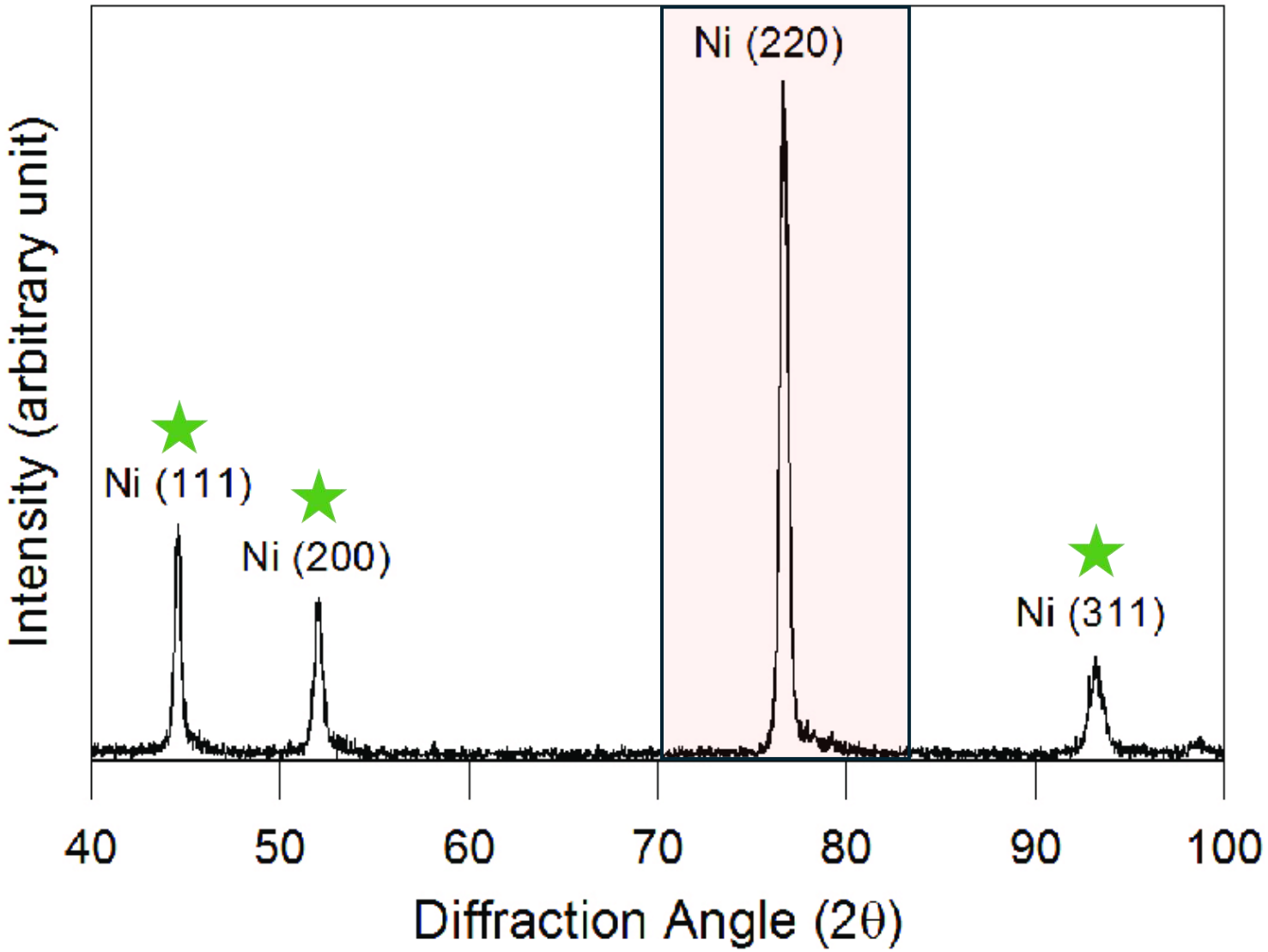


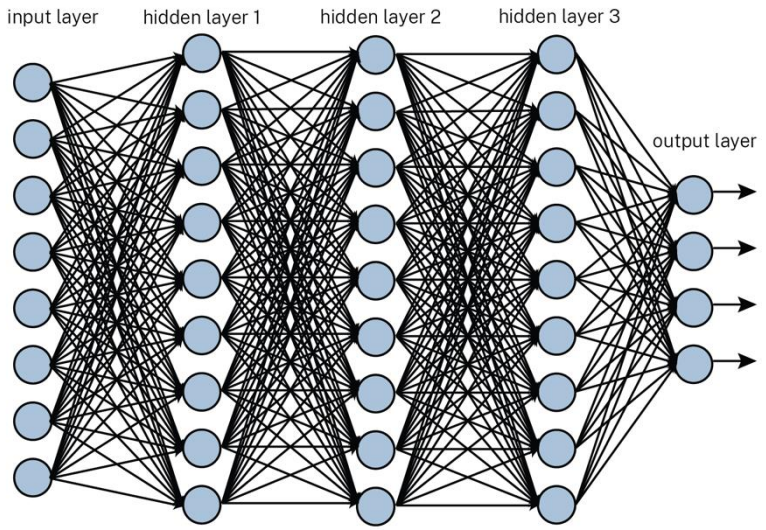


Dropout discourages reliance on any single feature by making it temporarily unavailable during training.



Why dropout helps, conceptually





Dropout can help us not overly rely on the largest peak, forcing it to also use smaller peaks

Why dropout helps, conceptually

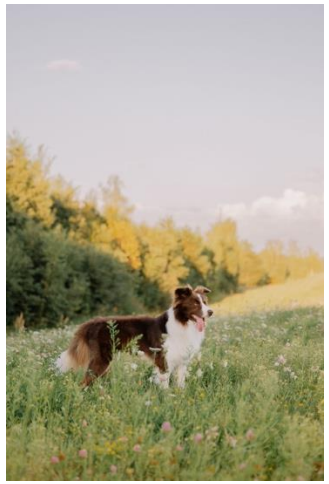
Wolves






Dogs

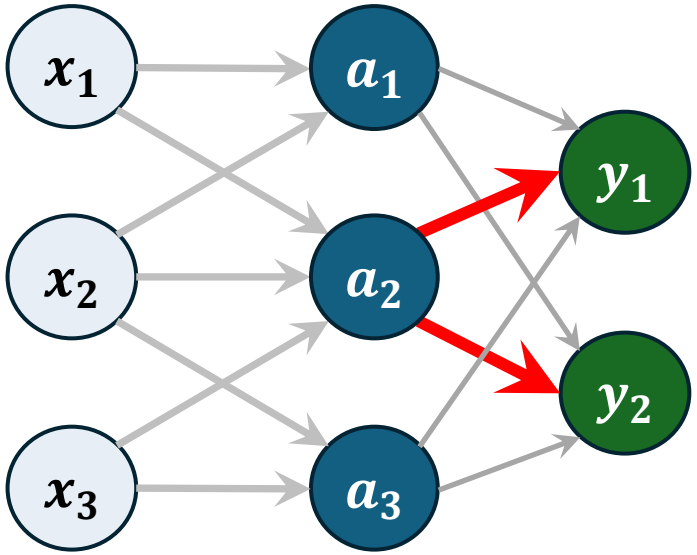





Without regularization, our model picks the "easiest" route toward low training loss; in this case: **Snowy background = wolf** **Grassy background = dog**

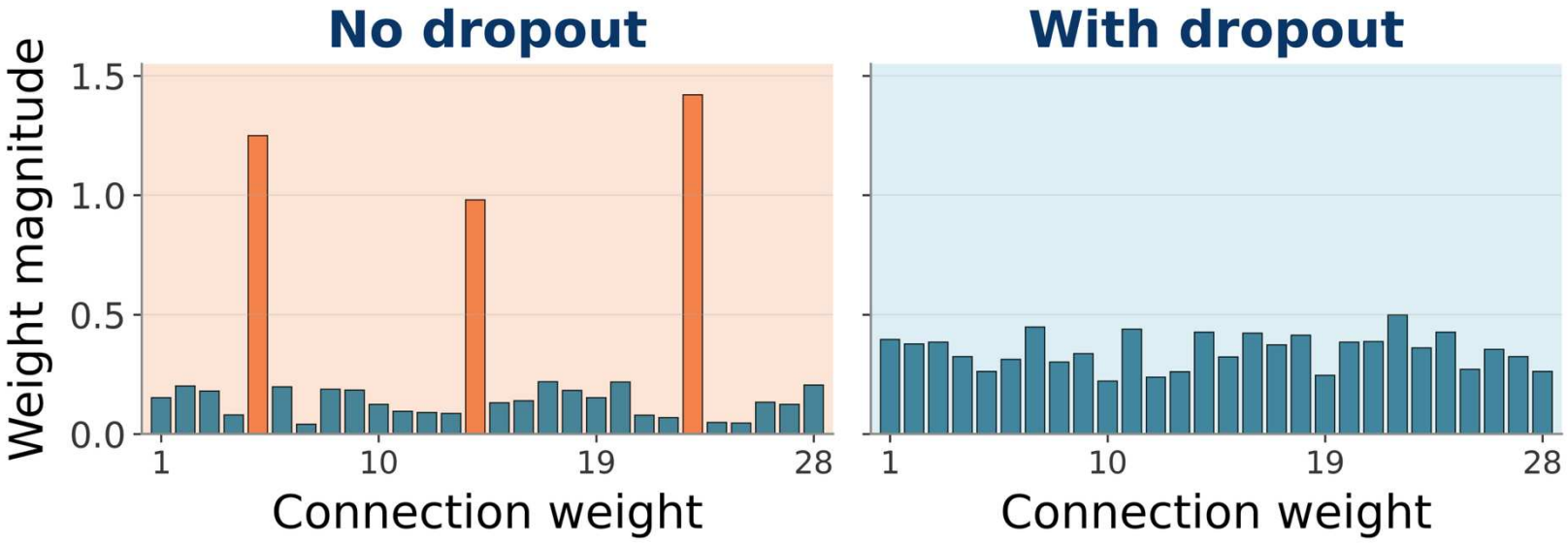
Dropout helps prevent this. Suppose the background were "dropped out" 50% of the time during training – our models would have to learn from the animal itself

Why dropout helps, conceptually

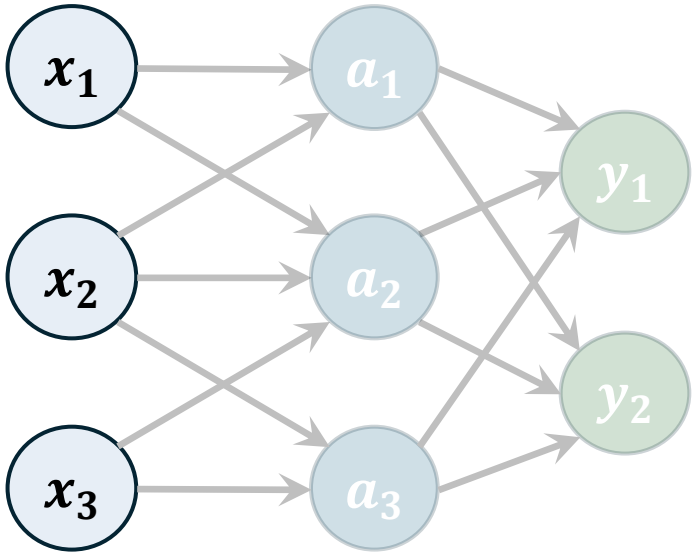


- The same idea holds for connections between hidden layers – we don't want the model to rely too much on specific activations
- Remember that hidden neurons are essentially *learned features* of the data

Dropout encourages **distributed** weights. Useful information is **not concentrated** in just one hidden unit.

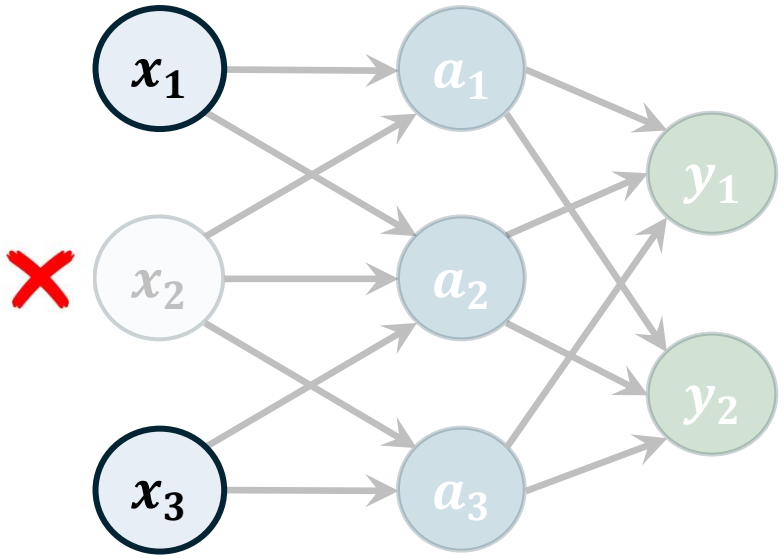


Dropout in practice: Step-by-step



Feed input data (x_i) to the neural network

Dropout in practice: Step-by-step



Feed input data (x_i) to the neural network

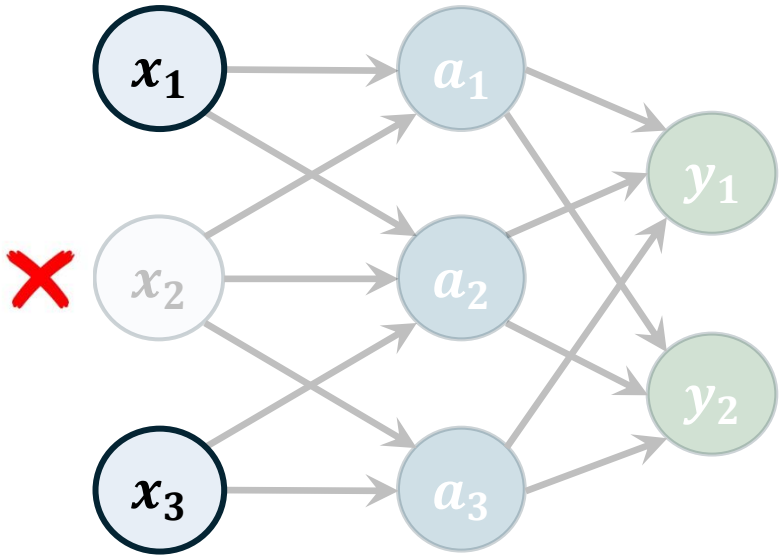
↓

Randomly set $x_i^{(0)} = 0$ with probability $p^{(0)}$

$$x_i^{(0)} = \begin{cases} 0, \text{ with probability } p^{(0)} \\ x_i^{(0)}, \text{ with probability } 1 - p^{(0)} \end{cases}$$

Where p is called the **dropout rate.**
For the input layer, this is usually quite small: p~0.05 to 0.2

Dropout in practice: Step-by-step



Feed input data (x_i) to the neural network

↓

Randomly set $x_i^{(0)} = 0$ with probability $p^{(0)}$

↓

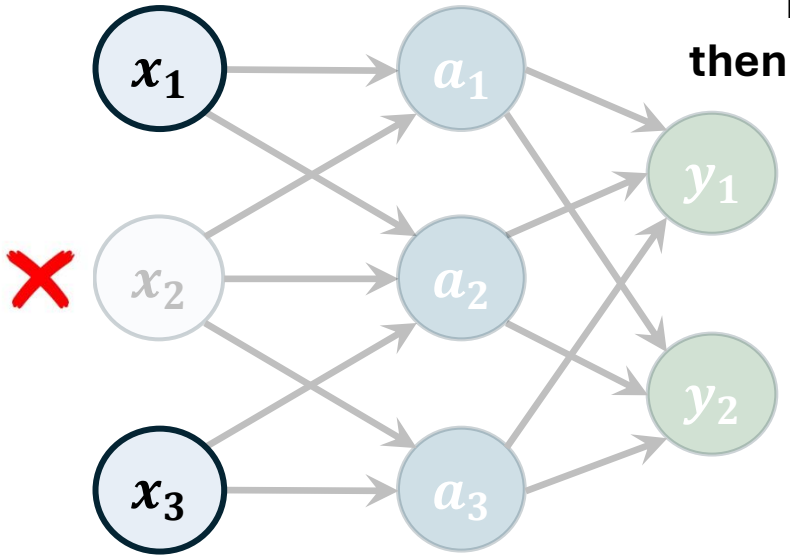
Rescale the values that were not dropped out

$$\tilde{x}^0{}_i = \begin{cases} 0, \text{ with probability } p^{(0)} \\ \dfrac{x_i^{(0)}}{1-p}, \text{ with probability } 1-p^{(0)} \end{cases}$$

Divide by $1-p$, the portion of weights kept

Scaling keeps the expected **activation magnitude the same during training and inference**, despite randomly setting some activations to zero.

Dropout in practice: Step-by-step



This is activated, then fed to next layer

$$z_i^{(1)} = w_{i1}^{(0)} x_1 + w_{i2}^{(0)} x_2 + w_{i3}^{(0)} x_3 + b_i$$

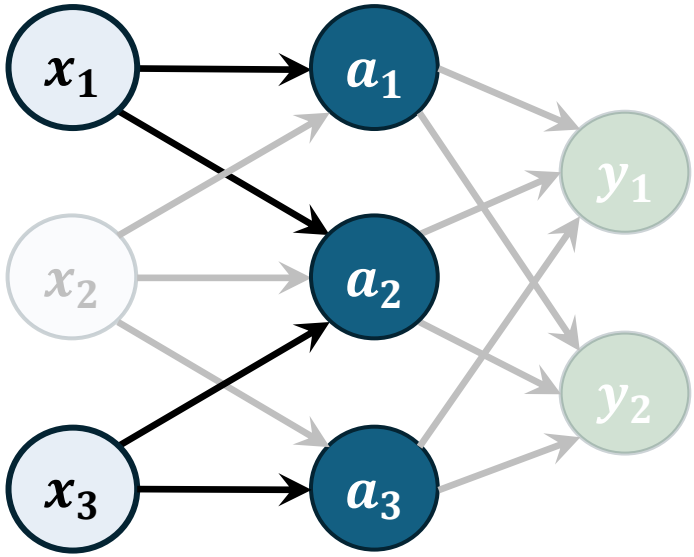
If we randomly drop a term from this sum, the **input to the next layer** is reduced during training. Hence the need for rescaling.

$$\tilde{x}^0{}_i = \begin{cases} 0, \text{ with probability } p^{(0)} \\ \dfrac{x_i^{(0)}}{1-p}, \text{ with probability } 1-p^{(0)} \end{cases}$$

Divide by $1-p$, the portion of weights kept

Scaling keeps the expected **activation magnitude the same during training and inference**, despite randomly setting some activations to zero.

Dropout in practice: Step-by-step



Feed input data (x_i) to the neural network

Randomly set $x_i^{(0)} = 0$ with probability $p^{(0)}$

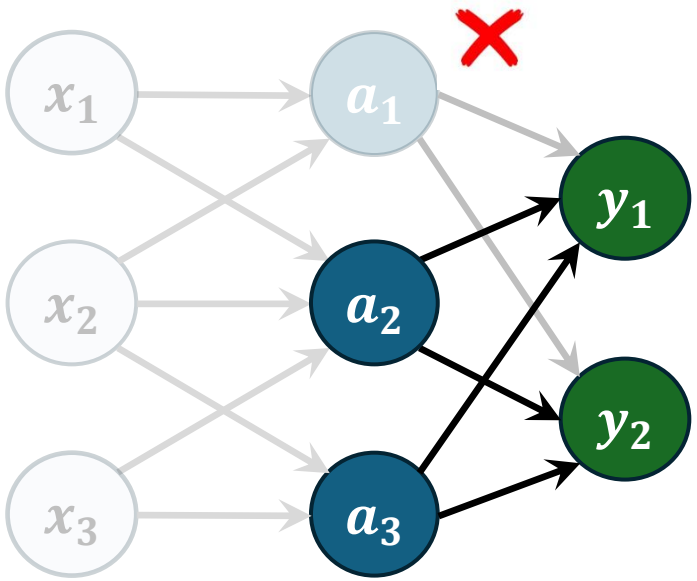
Rescale the values that were not dropped out

Pass the modified input to the next layer

$$z_i^{(1)} = w_{i1}^{(0)} x_1 + w_{i2}^{(0)} x_2 + w_{i3}^{(0)} x_3 + b_i$$

$$a_i^{(1)} = \text{Activation}(z_i^{(1)})$$

Dropout in practice: Step-by-step



Same procedure as before!

Except now we set activations (not inputs) = 0

And hidden layer dropout is usually higher: p~0.2 to 0.5

Feed input data (x_i) to the neural network

↓

Randomly set $x_i^{(0)} = 0$ with probability $p^{(0)}$

↓

Rescale the values that were not dropped out

↓

Pass the modified input to the next layer

↓

Randomly set $a_i^{(l)} = 0$ with probability $p^{(l)}$

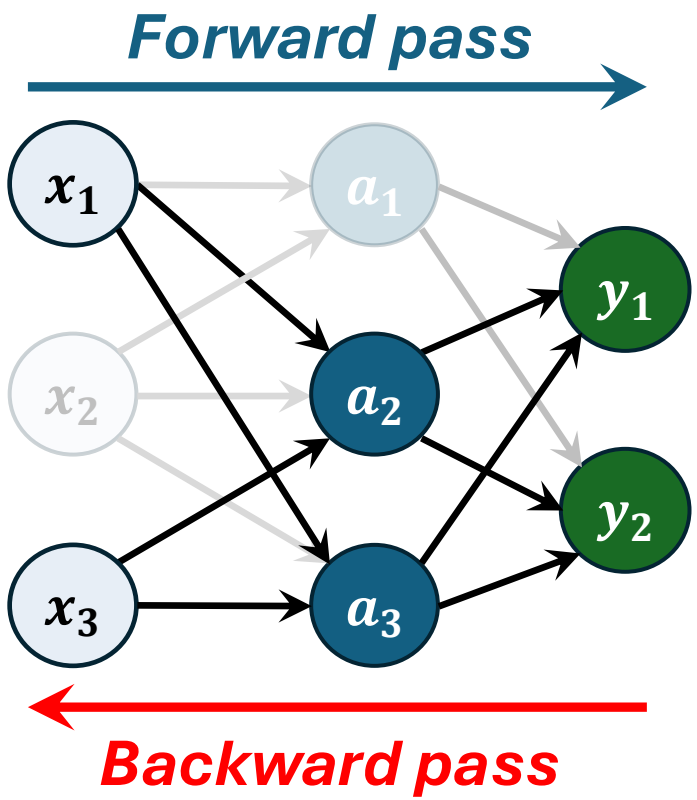
Then repeat until we reach the output layer

Dropout visualized



Remember that different activations will be set to zero on each forward pass.
So as we train over hundreds/thousands of epochs, **all activations still get used.**

Handling backprop with dropout



Recall: the derivative of the loss with respect to each weight

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}$$

This feeds into gradient descent

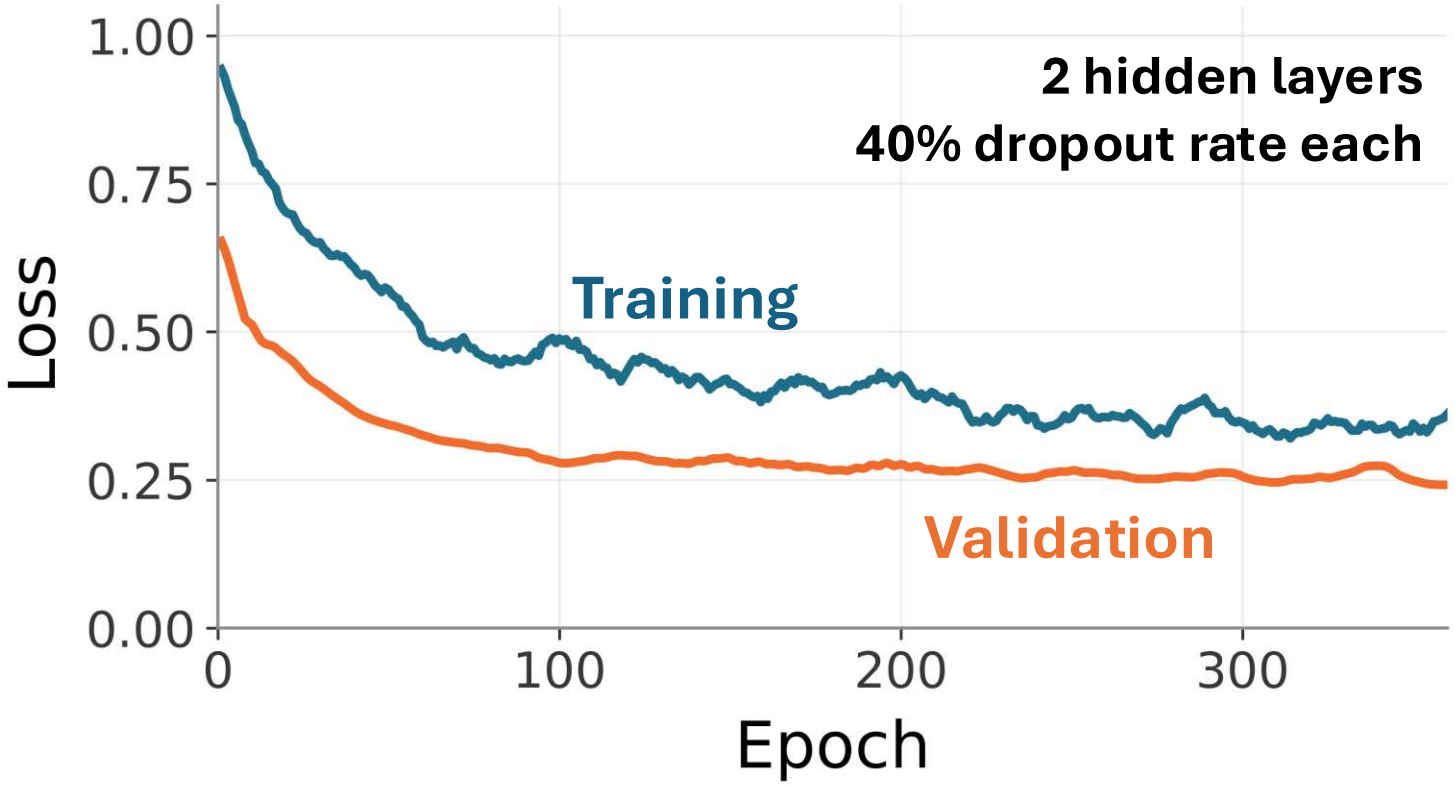
$$\text{New } w_{ji}^{(l)} = \text{Old } w_{ji}^{(l)} - \eta \left(\frac{d\mathcal{L}}{dw_{ji}^{(l)}} \right)$$

Same process with dropout, except now some $\tilde{a}_i^{(l-1)} = 0$ and their outgoing weights have **zero gradient** on that pass

Certain weights don't get updated! Preventing them from becoming large very fast.

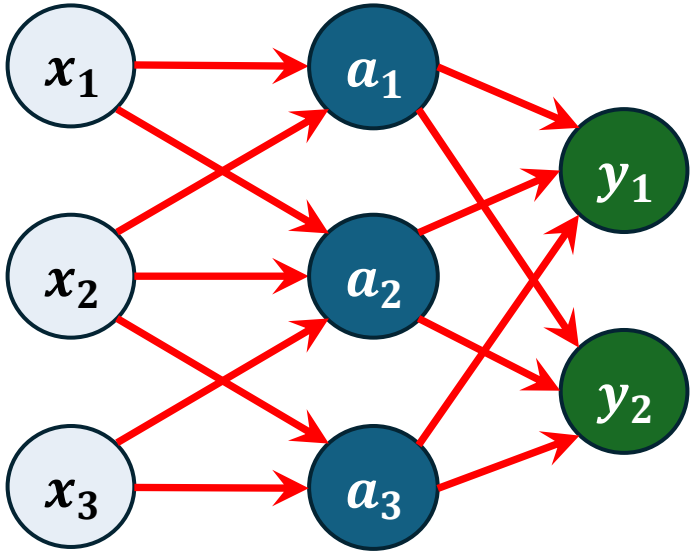
$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}} = 0 \qquad \text{New } w_{ji}^{(l)} = \text{Old } w_{ji}^{(l)}$$

Training with dropout



Two things to notice:

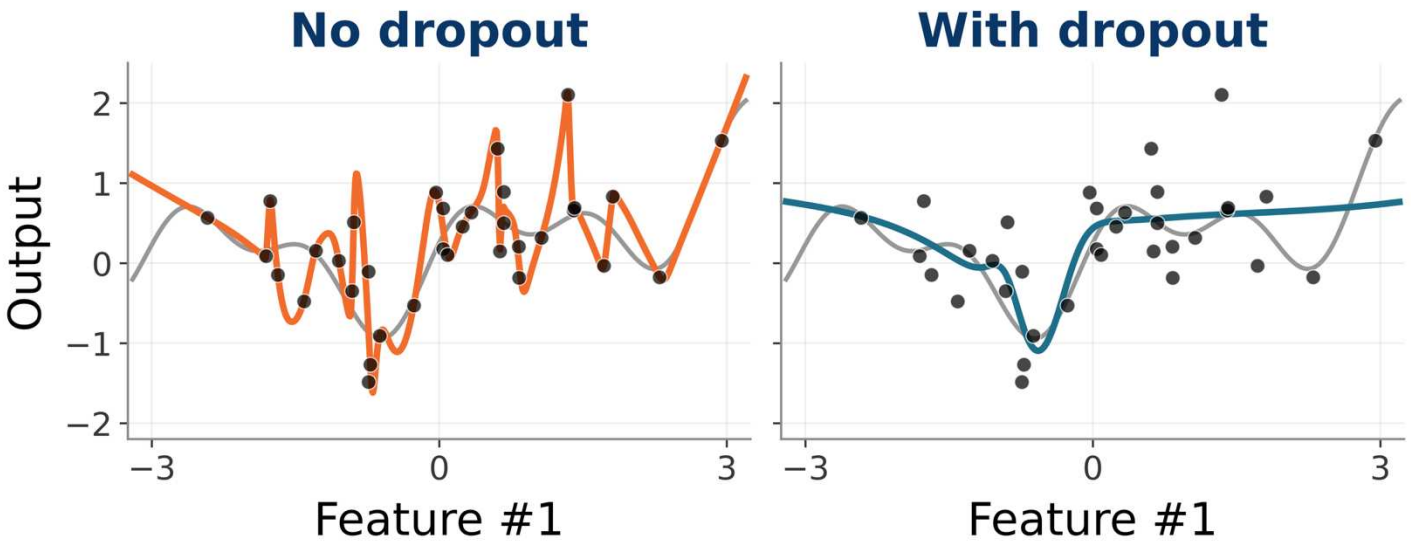
1. Even at many epochs, we don't overfit!
2. Val loss < training loss, since dropout is not used during validation



At inference (which includes validation) we do not set any activations equal to zero.

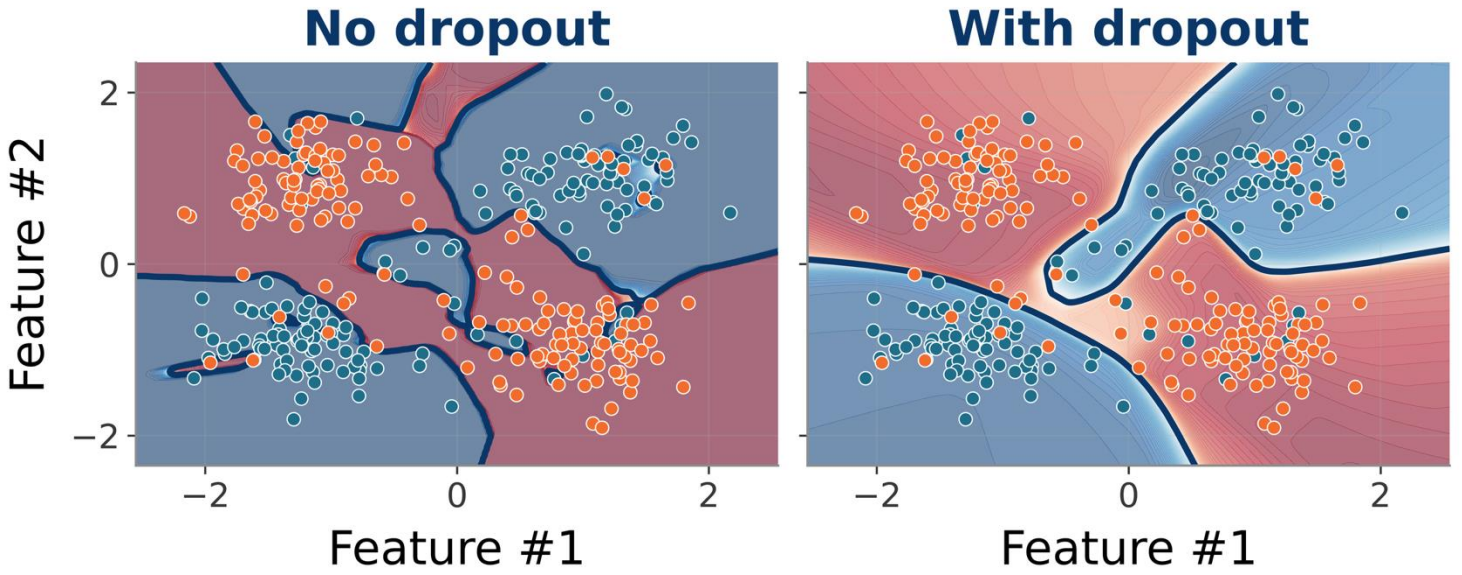
No dropout when making real predictions!

Some examples with and without dropout

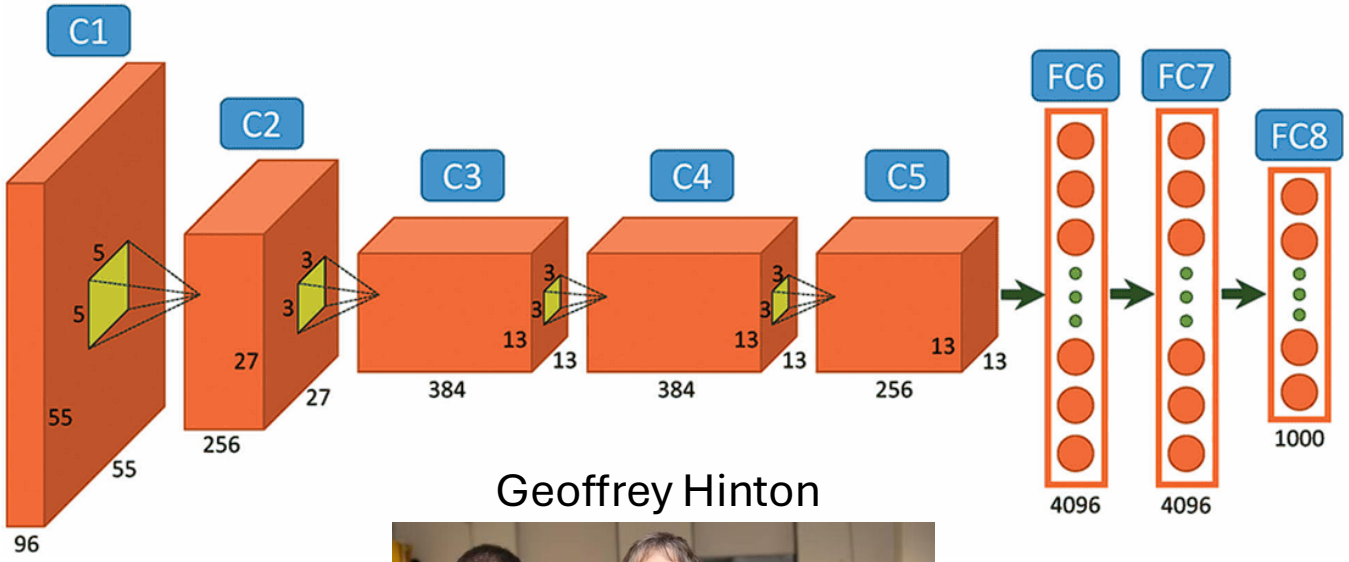


Results are similar to L2 regularization; We are much less prone to overfitting with dropout than without it

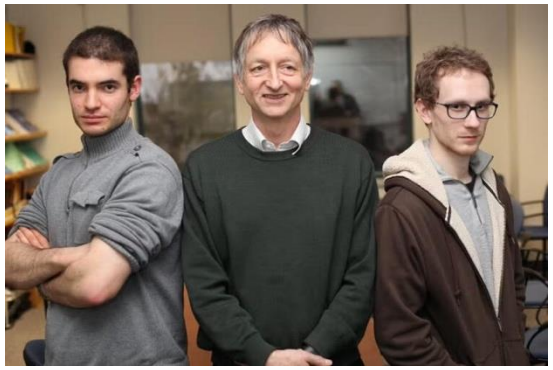
Unlike L2 regularization, dropout doesn't explicitly penalize large weights, so we can often still learn complex lines/boundaries



Dropout is widely used!



Geoffrey Hinton



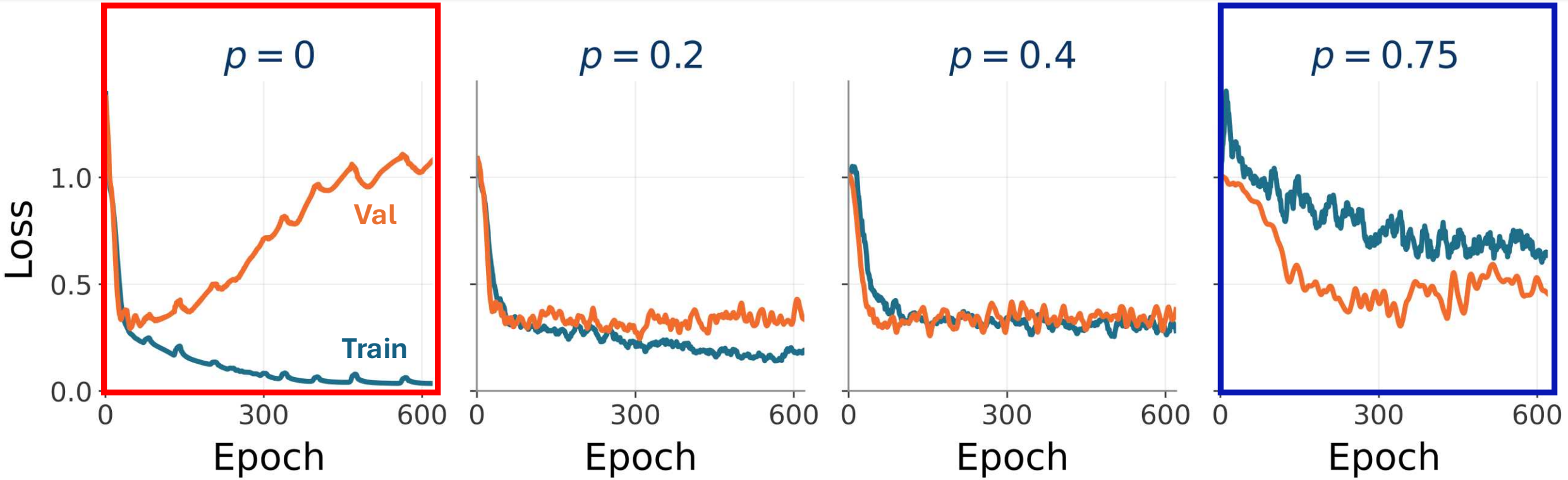
The Godfather (of AI)

Cool guys who do ML



AlexNet (2012) is a famous example of a CNN for image classification, which used a **dropout rate of 50%** between its fully connected layers

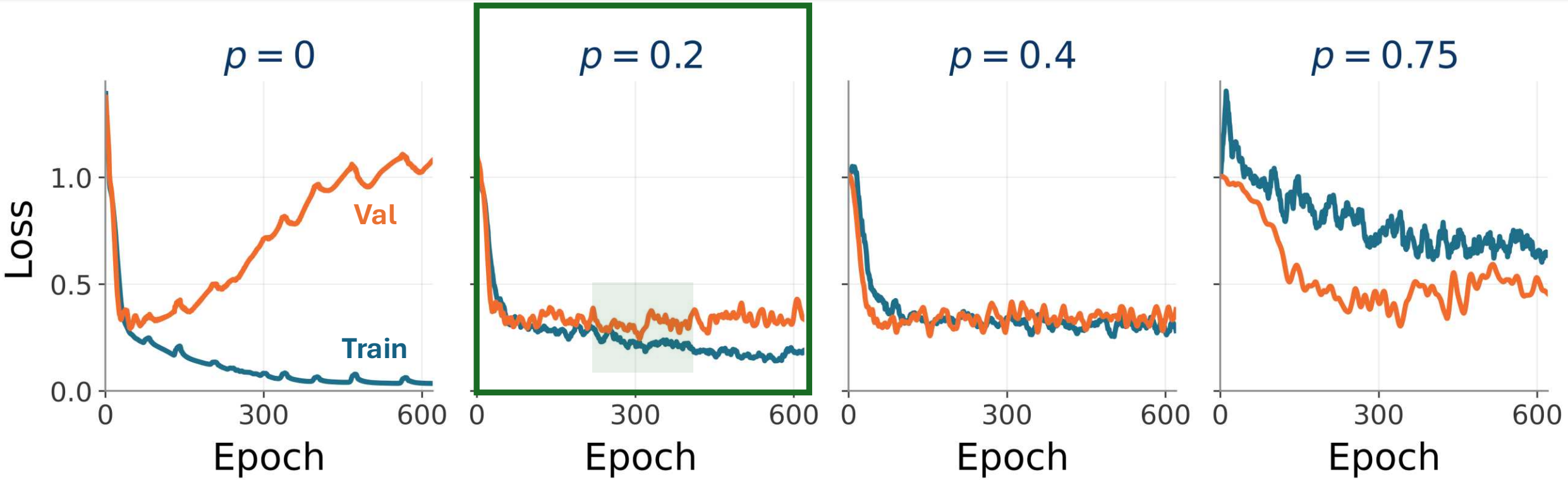
Effect of the dropout rate



Without any dropout, we overfit very quickly

With too much dropout, the loss becomes very noisy, and the model is forced to be overly simple

Effect of the dropout rate



The sweet spot is somewhere in the middle. **Dropout rate is a hyperparameter just like any other.** Try different values and see which gives the lowest validation loss (with early stopping).

Questions?

Regularization techniques covered so far…

Early stopping

You should basically always use this, often in combination with others

L1/L2 regularization

Use this to penalize large weights, or to permanently "zero out" some features

Dropout

Use this to prevent overfitting on specific features, while keeping all of them active

All three affect the model, but can we also prevent overfitting by **changing or augmenting the training data itself?**

More (unique) data = less likely to overfit



vs.



Suppose we train a neural network to **classify wolf *vs.* dog**

With just **2 training samples**, our model could learn to distinguish photos based on white *vs.* green background…Not what we want!

Remember, models take the "easiest" route to achieving low loss, not necessary the right one!

To prevent the model from cheating…give it more data (ideally, more diverse data where it can't cheat)



More (unique) data = less likely to overfit





vs.





Suppose we train a neural network to **classify wolf *vs.* dog**

With **4 training samples**, our model now cannot distinguish photos based on white *vs.* green background, as that would lead to high loss. But now it could just distinguish based on the **color of the animal**

Let's keep adding data…

More (unique) data = less likely to overfit







vs.







Suppose we train a neural network to **classify wolf *vs.* dog**

With **6 training samples**, our model now cannot distinguish using background, nor on animal color alone, but realistically this model still likely would not generalize well….

For example, what if we threw a yorkie at it?



But in many cases, we don't have more data

In general: more data is better – but of course, data collection is expensive
If we are unable to collect *new* data, how about we **modify our existing data?**





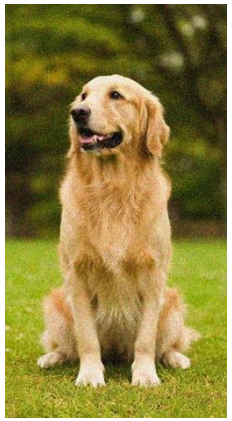
Color filtering



Cropping



Rotation



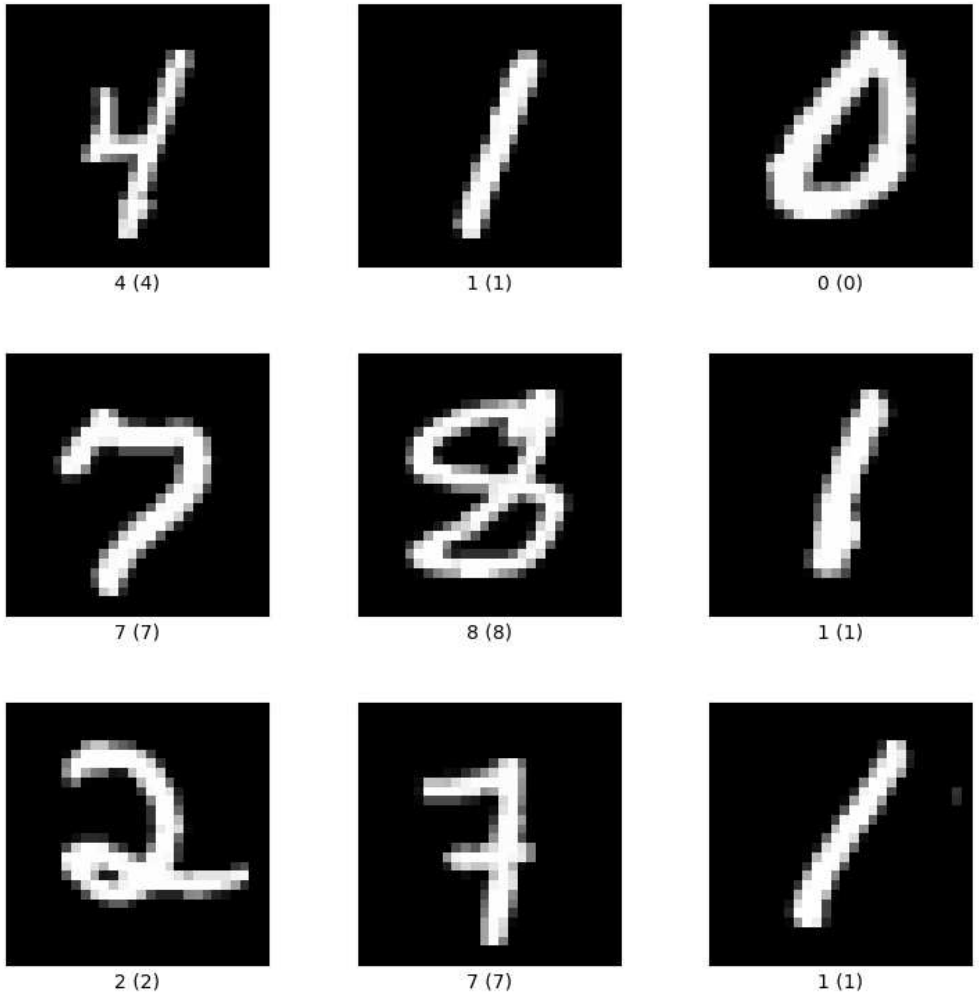
Noise

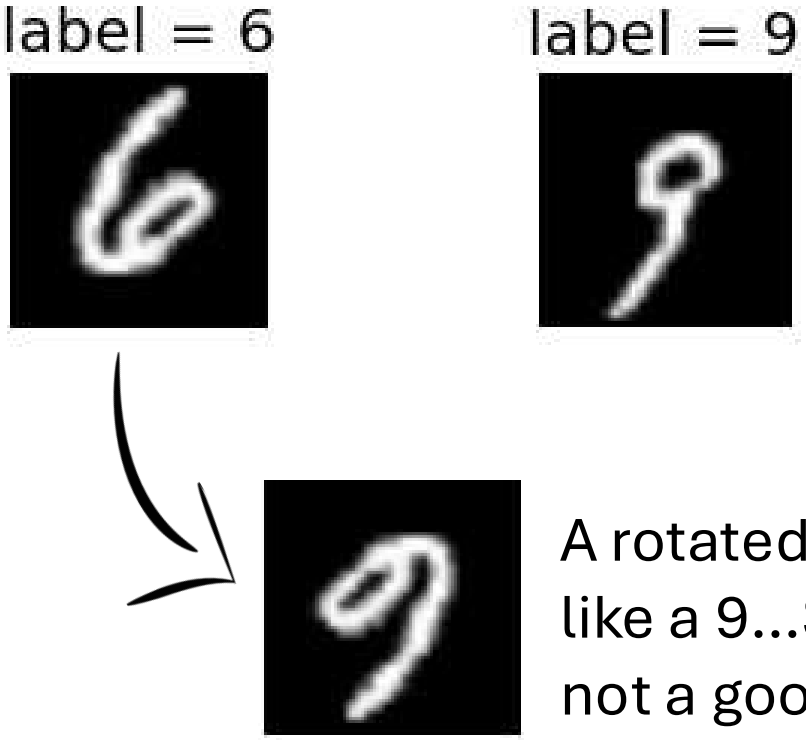
The label (dog) unchanged. We get "new" labeled samples for free!

...And many more methods...

This is known as ***data augmentation.***
A very effective and common approach!

Use with caution…Don’t change too much!





A rotated 6 looks like a 9…So this is not a good form of augmentation!

Other examples of data augmentation

Data type	Augmentation methods
Images	Crop, resize, flip, rotate, noise
Audio	Time shift, noise, pitch shift, frequency masking
Text	Synonym replacement, word deletion, paraphrasing
Time series	Noise, window cropping, scaling, segment permutation
Tabular	Feature noise/dropout, interpolation between known samples

Other examples of data augmentation

Data type	Augmentation methods
Images	Crop, resize, flip, rotate, noise
Audio	Time shift, noise, pitch shift, frequency masking
Text	Synonym replacement, word deletion, paraphrasing
Time series	Noise, window cropping, scaling, segment permutation
Tabular	Feature noise/dropout, interpolation between known samples

In summary…

Early stopping

You should basically always use this, often in combination with others

L1/L2 regularization

Use this to penalize large weights, or to permanently "zero out" some features

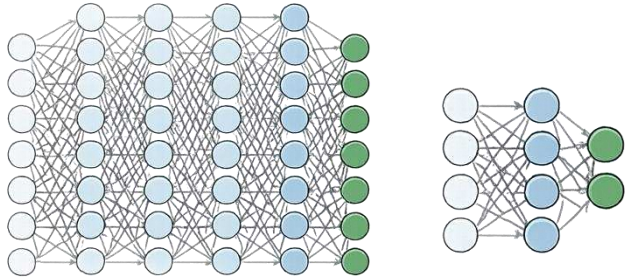
Dropout

Use this to prevent overfitting on specific features, while keeping all of them active

Data augmentation

Use this when you have limited data, and when modifying the data does *not* change the label

Try these all in various combinations! See which gives the lowest validation loss. When all else fails, and you keep overfitting…**Use a simpler model**



With less neurons, we are much less likely to overfit

Questions?

What to know for the final exam

- Know **how each dropout technique differs**, conceptually
  - Early stopping, L1 and L2, dropout, data augmentation
- You should be able to **do the math on L1/L2 and dropout**
- Understand how to **control the strength** of regularization, and its effect on 1) training/validation loss, and 2) over/underfitting
- For **data augmentation**: What changes are allowed