Deep Learning

Ulf Brefeld & Soham Majumder

build: May 28, 2024

Machine Learning Group
Leuphana University of Lüneburg

Regularization

At first, we have motivated the use of a loss function with a Bayesian formulation combining the probability of the data given the model and the probability of the model:

$$\log \mu_W(w|\mathcal{D} = \mathbf{d}) = \log \mu_{\mathcal{D}}(\mathbf{d}|W = w) + \log \mu_W(w) - \log Z.$$

If μ_W follows a multivariate Gaussian distribution, we have

$$\mu_W(w) = \frac{1}{Z'} \exp\left(-\frac{1}{2}(w-\mu)^\top \Sigma^{-1}(w-\mu)\right).$$

Now, further assume this Gaussian has variance $\Sigma = \frac{1}{2\lambda}\mathbb{1}$ and zero mean. Then,

$$\begin{aligned}\mu_W(w) &= \frac{1}{Z'}\exp\left(-\frac{1}{2}(w-\mu)^\top\Sigma^{-1}(w-\mu)\right)\\ &= \frac{1}{Z'}\exp\left(-\lambda w^\top w\right)\\ &= \frac{1}{Z'}\exp\left(-\lambda\|w\|_2^2\right),\end{aligned}$$

and thus $\log\mu_W(w) = -\lambda\|w\|_2^2 - \log Z'$. Since $\log Z'$ does not affect which parameters are optimal, we can disregard it.

We usually write our optimization problems as minimization. Therefore, we minimize $-\log \mu_W(w)$, which yields the convex penalty

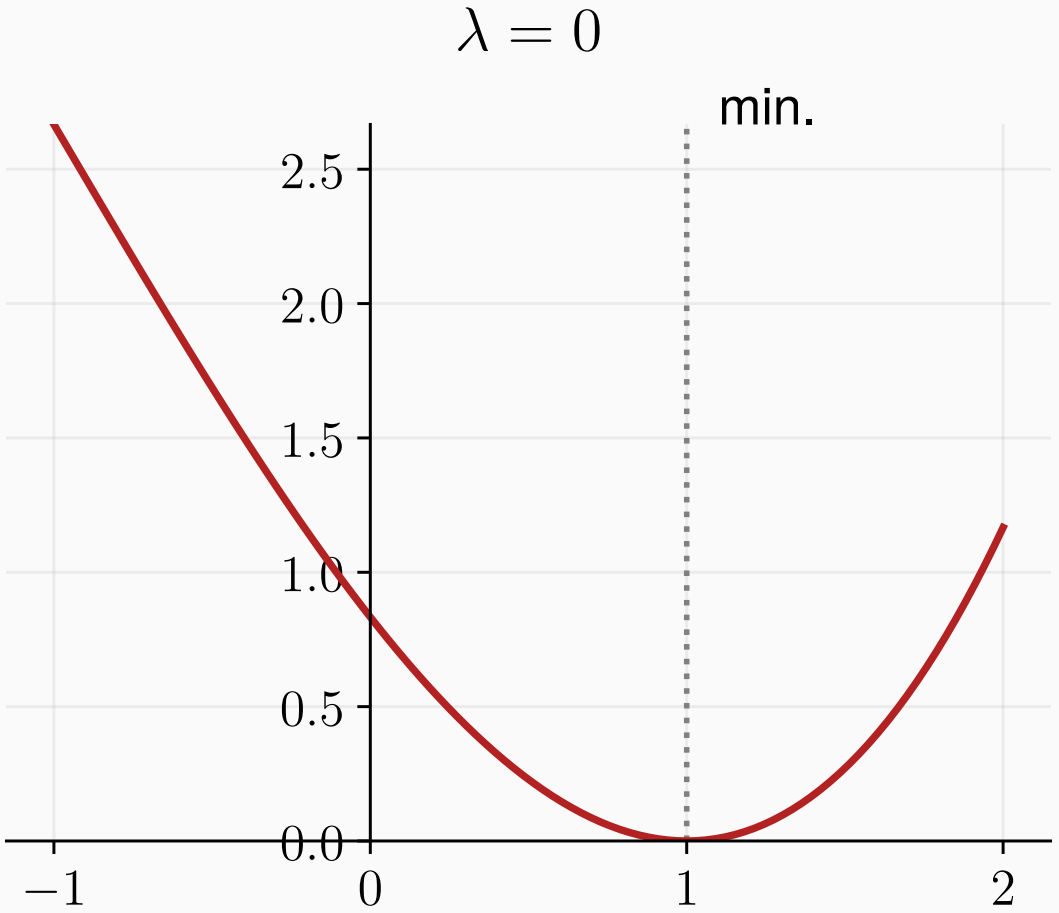
$$\lambda \|w\|_2^2$$

on the parameters w.

This is called the $\boldsymbol{\ell_2}$**-regularization**, or *weight decay*.

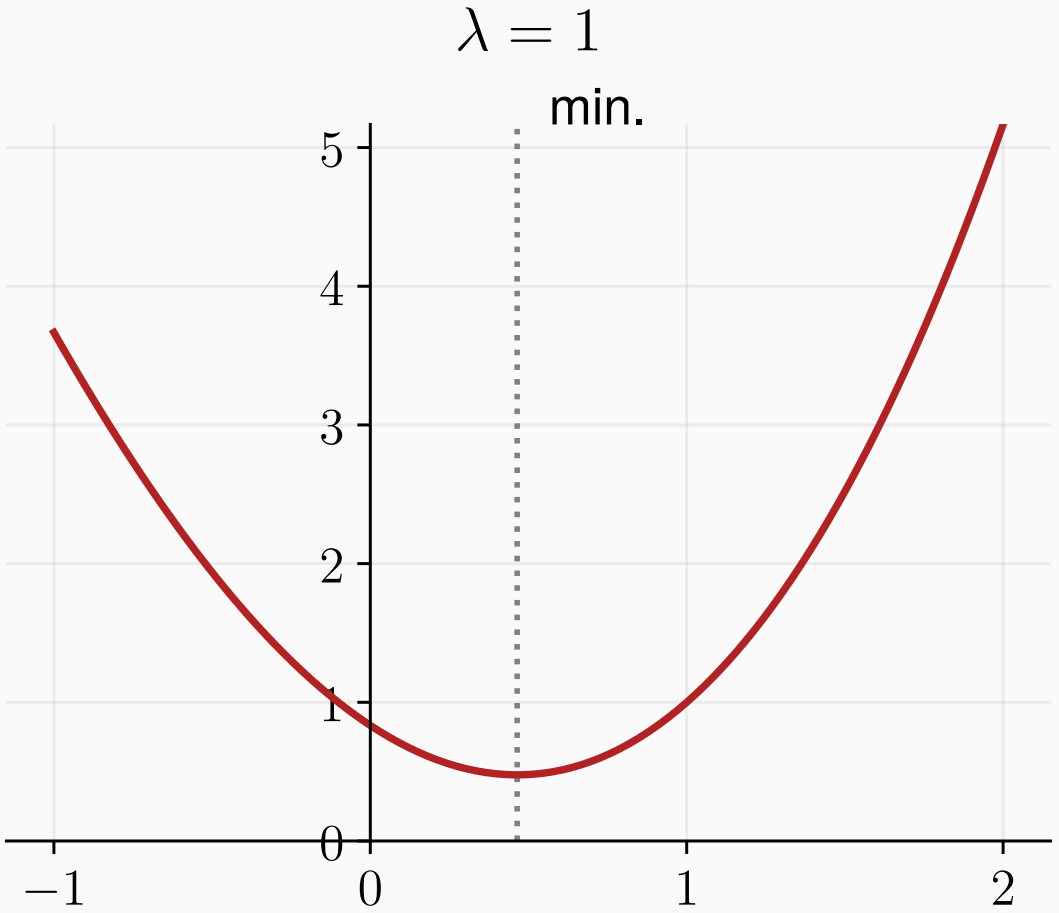
Increasing λ moves the optimum closer to 0 and away from the optimum for the loss alone.

$$E(w) = (w-1)^2 + \frac{1}{6}(w-1)^3 + \lambda \|w\|_2^2$$



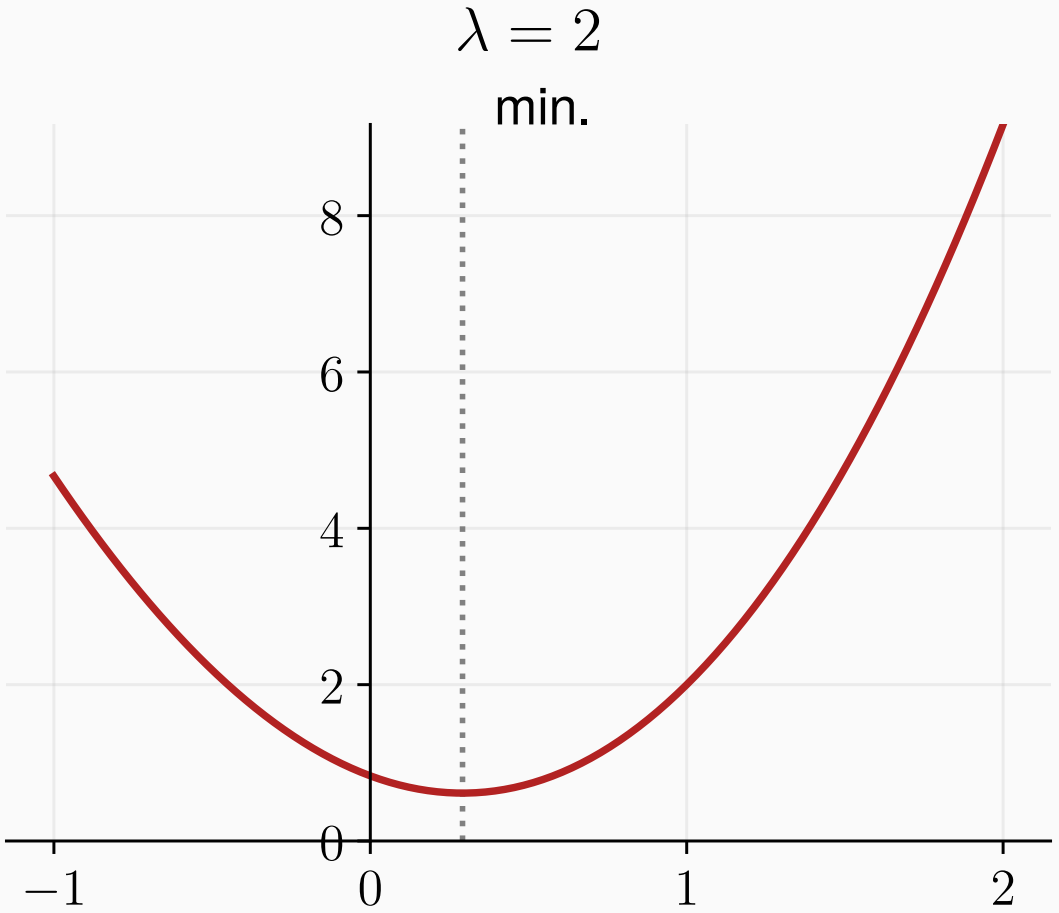
Increasing λ moves the optimum closer to 0 and away from the optimum for the loss alone.

$$E(w) = (w-1)^2 + \frac{1}{6}(w-1)^3 + \lambda \|w\|_2^2$$



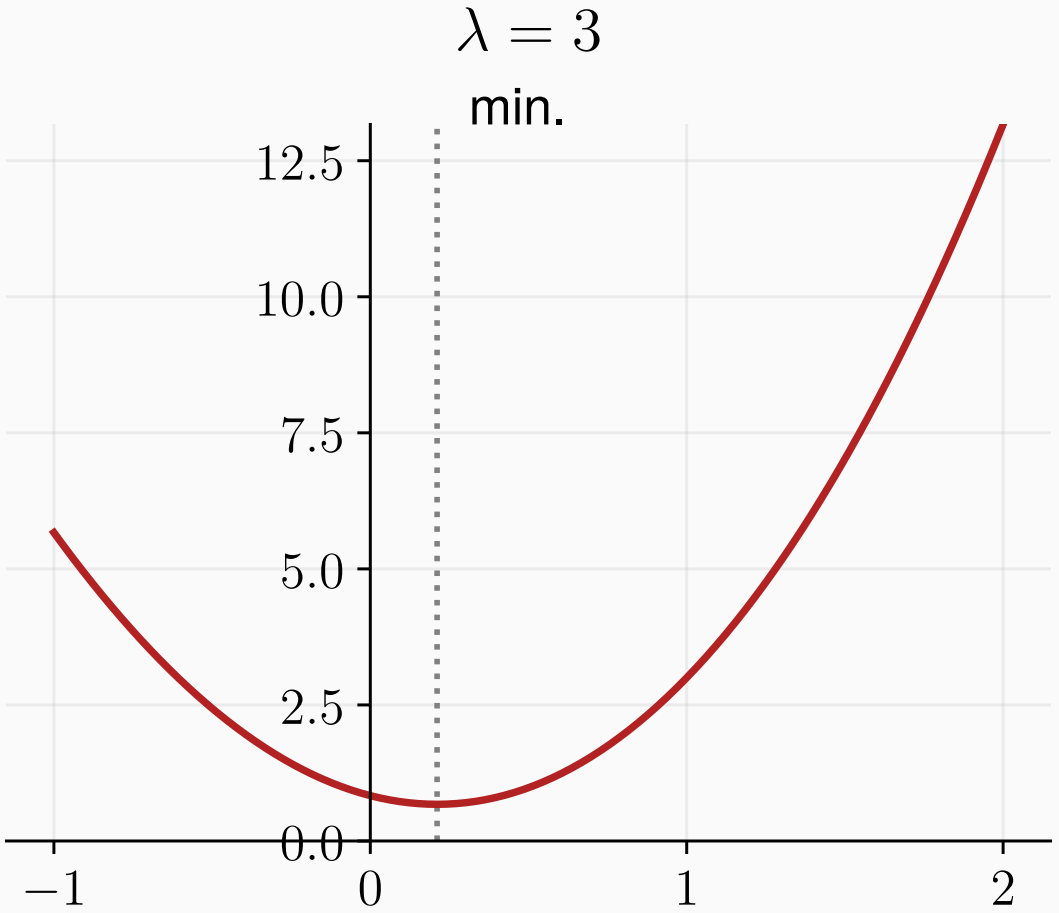
Increasing λ moves the optimum closer to 0 and away from the optimum for the loss alone.

$$E(w) = (w-1)^2 + \frac{1}{6}(w-1)^3 + \lambda \|w\|_2^2$$



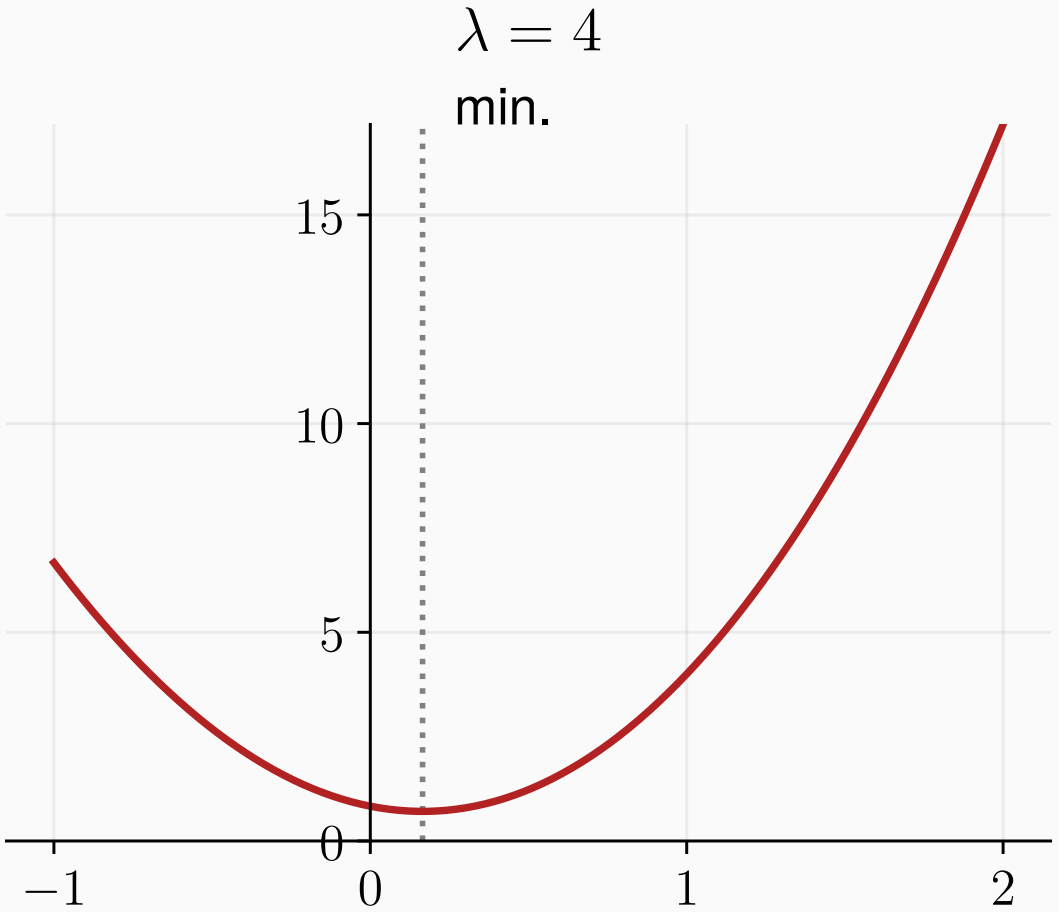
Increasing λ moves the optimum closer to 0 and away from the optimum for the loss alone.

$$E(w) = (w-1)^2 + \frac{1}{6}(w-1)^3 + \lambda \|w\|_2^2$$



Increasing λ moves the optimum closer to 0 and away from the optimum for the loss alone.

$$E(w) = (w-1)^2 + \frac{1}{6}(w-1)^3 + \lambda \|w\|_2^2$$



We can apply the exact same scheme with a Laplace prior

$$\begin{aligned}\mu(w) &= \frac{1}{(2b)^D} \exp\left(-\frac{1}{b}\sum_{d=1}^{D}|w|\right) \\ &= \frac{1}{(2b)^D} \exp\left(-\frac{\|w\|_1}{b}\right)\end{aligned}$$

which results in a penalty of the form

$$\lambda\|w\|_1$$

and is known as the ℓ_1**-regularization**.

Similar to its ℓ_2 counterpart, the penalty is convex.

An important property of the ℓ_1 regularization is that if E is convex and

$$w^* = \operatorname*{argmin}_w E(w) + \lambda \|w\|_1$$

then

$$\forall d, \left| \frac{\partial E}{\partial w_d} \right| < \lambda \Rightarrow w_d^* = 0$$

In practice it means that this penalty pushes some of the variables to zero, but in contrast to the ℓ_2 penalty, they remain there. ℓ_1 regularization requires the optimal solution to lie on a simplex

The λ parameter controls the sparsity of the solution.

With the ℓ_1 regularization, the update rule becomes

$$w_{t+1} = w_t - \eta g_t - \lambda \operatorname{sign}(w_t)$$

where sign is applied per component. This is almost identical to

$$w'_t = w_t - \eta g_t$$
$$w_{t+1} = w'_t - \lambda \operatorname{sign}(w'_t)$$

This update may overshoot and result in a component of w'_t strictly on one side of 0 while the same component in w_{t+1} is strictly on the other

While this is not a problem in principle, since w_t will fluctuate around zero, it can be an issue if the zeroed weights are handled in a specific manner (e.g. sparse coding to reduce memory footprint or computation).

The **proximal operator** takes care of preventing parameters from 'crossing zero', by adapting λ when it is too large

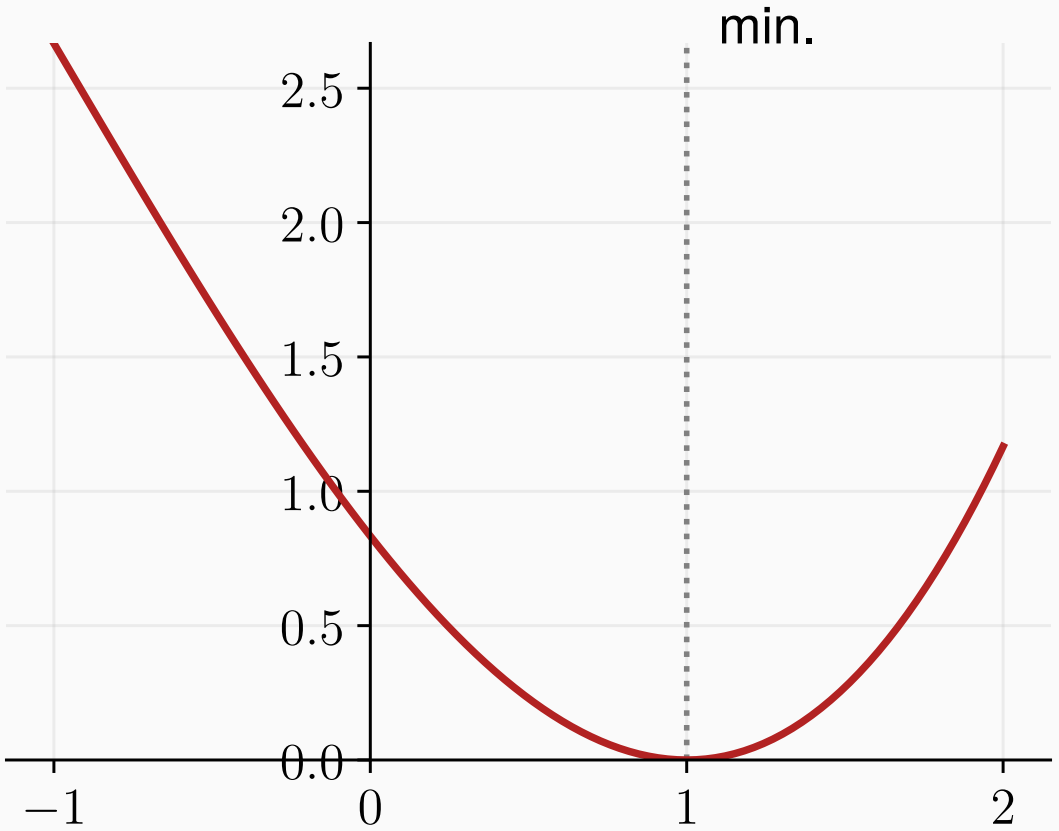
$$w'_t = w_t - \eta g_t$$
$$w_{t+1} = w'_t - \min(\lambda, |w'_t|) \odot \operatorname{sign}(w'_t)$$

where $\min$ is component-wise and $\odot$ is the Hadamard component-wise product

Increasing λ moves the optimum closer to 0 and away from the optimum for the loss alone.

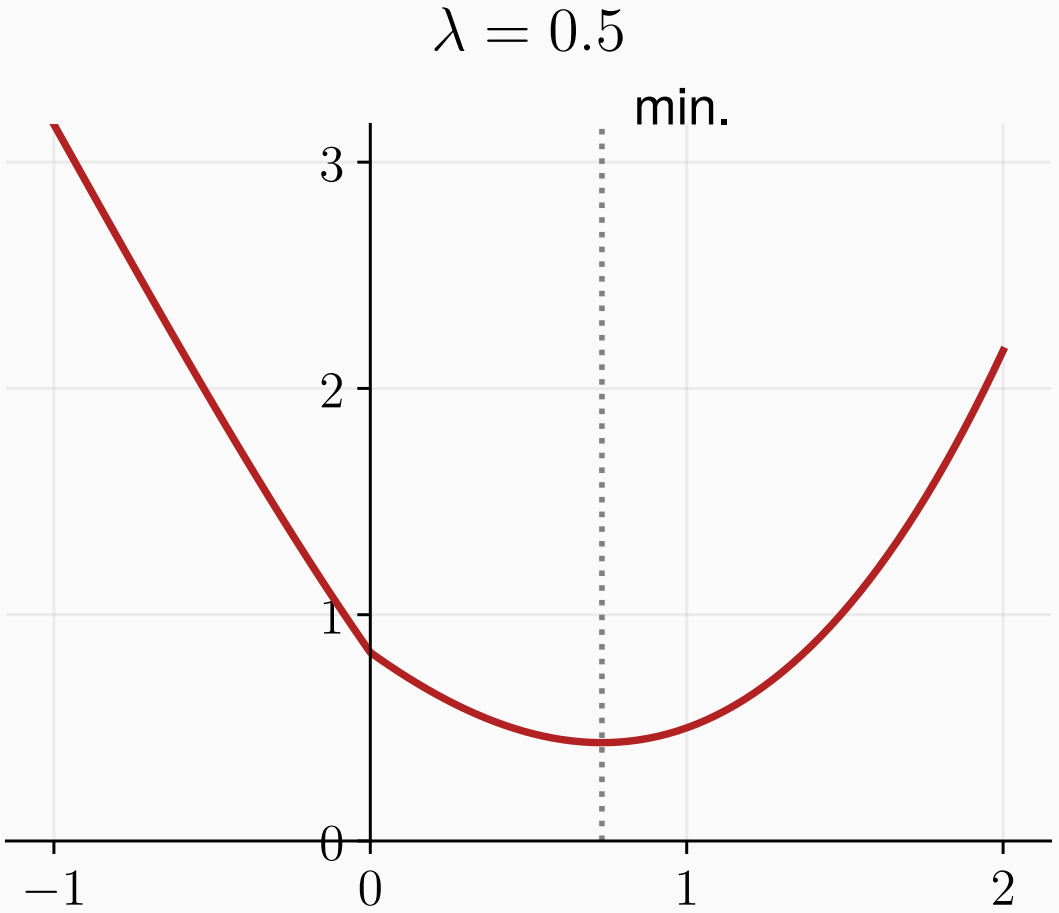
$$E(w) = (w-1)^2 + \frac{1}{6}(w-1)^3 + \lambda \|w\|_1$$

$\lambda = 0.0$



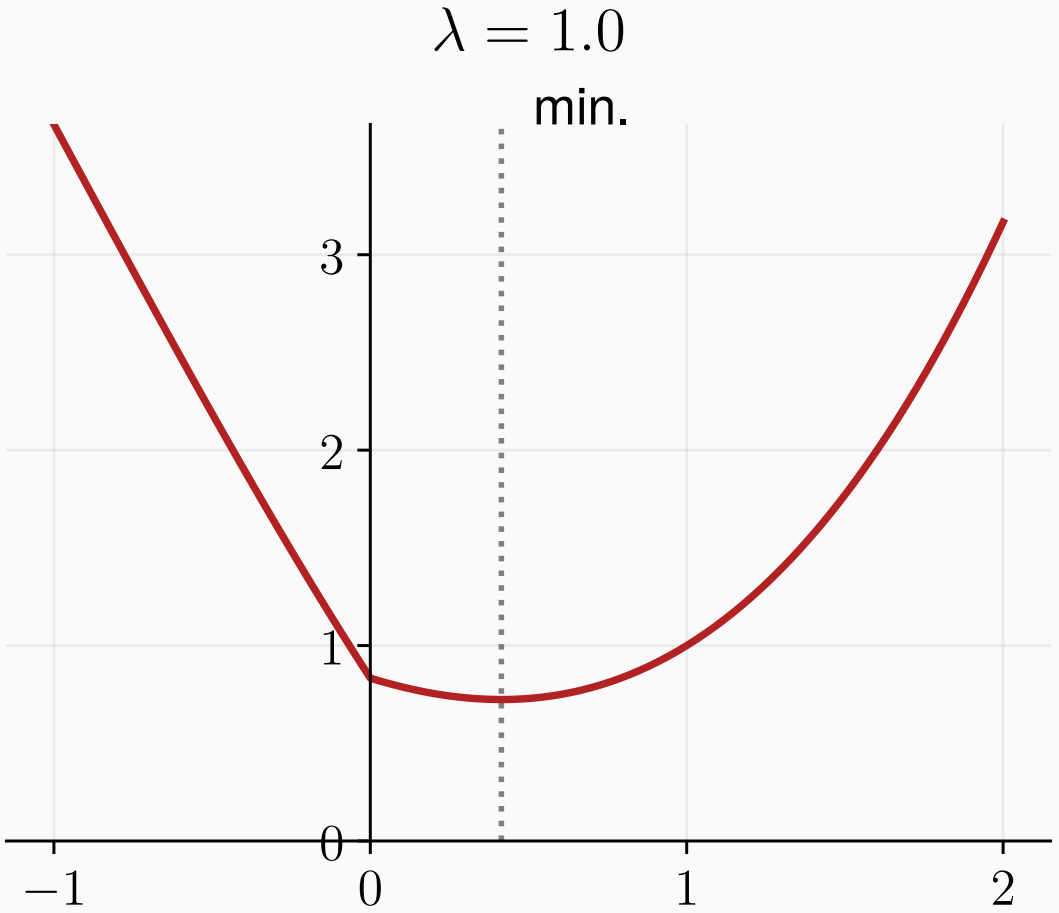
Increasing λ moves the optimum closer to 0 and away from the optimum for the loss alone.

$$E(w) = (w-1)^2 + \frac{1}{6}(w-1)^3 + \lambda \|w\|_1$$



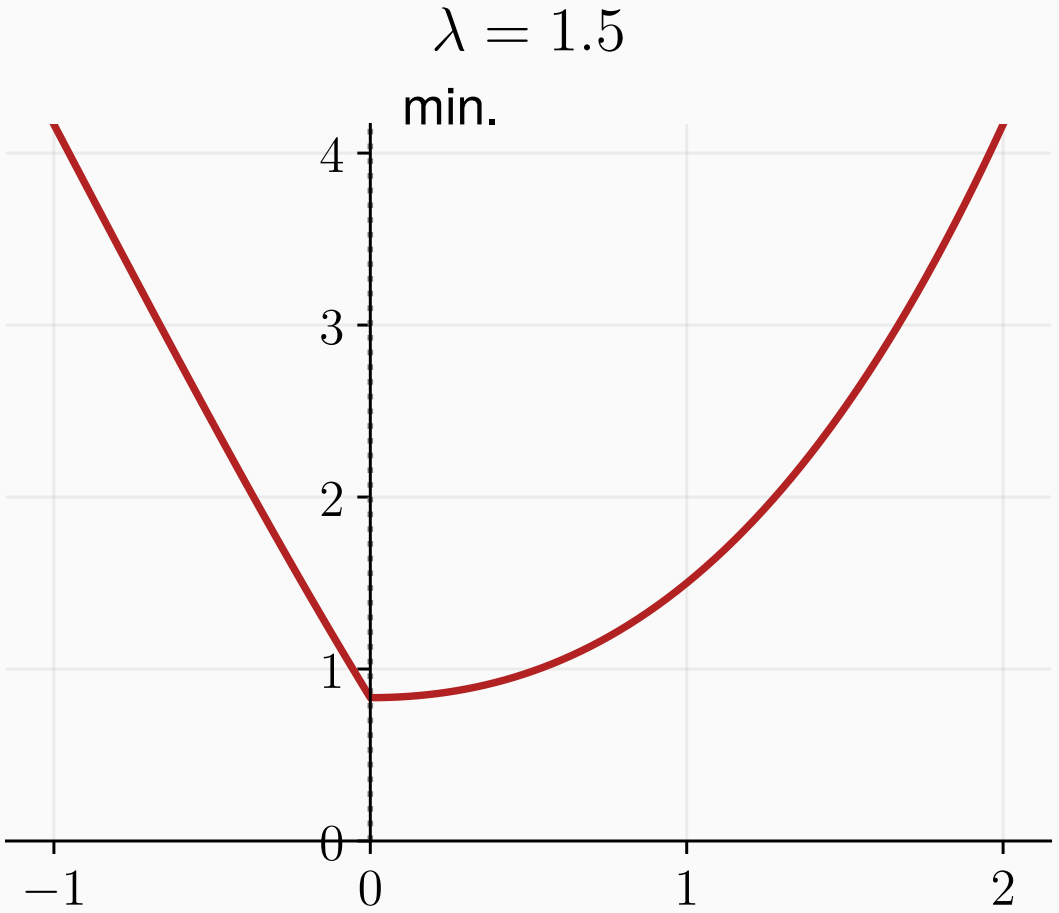
Increasing λ moves the optimum closer to 0 and away from the optimum for the loss alone.

$$E(w) = (w-1)^2 + \frac{1}{6}(w-1)^3 + \lambda \|w\|_1$$



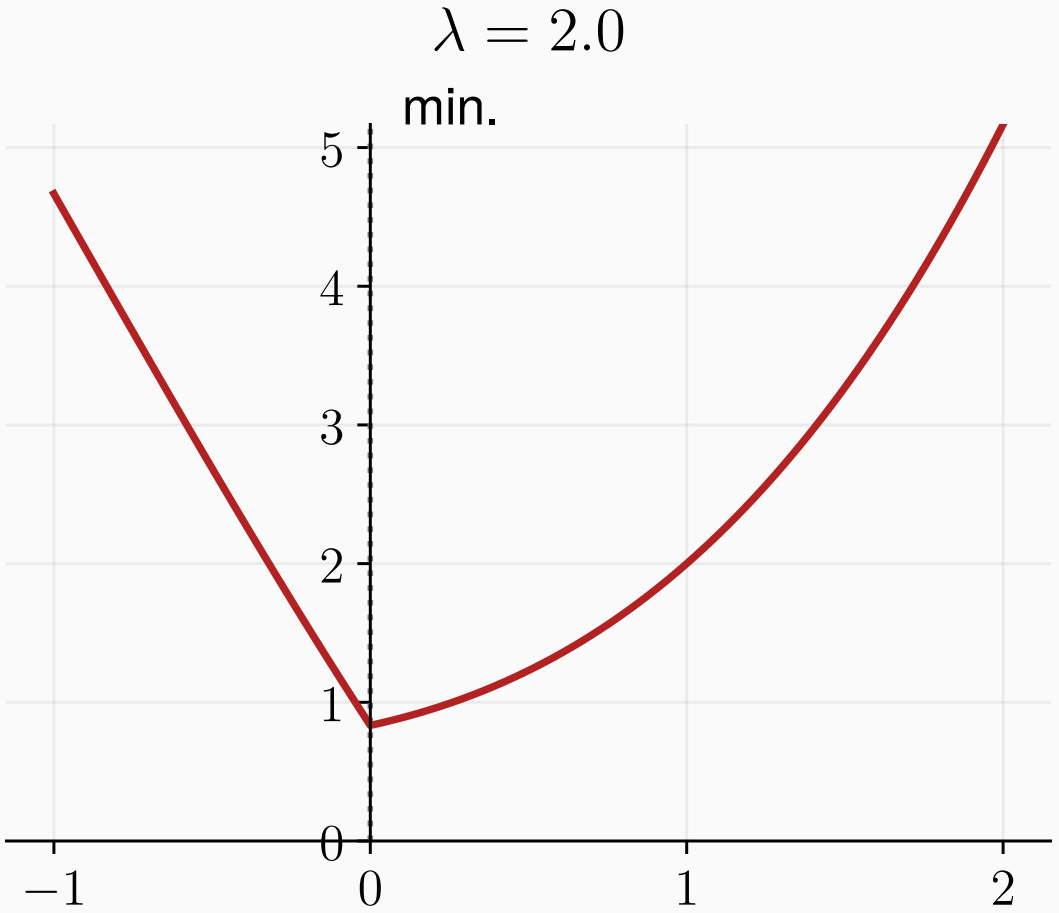
Increasing λ moves the optimum closer to 0 and away from the optimum for the loss alone.

$$E(w) = (w-1)^2 + \frac{1}{6}(w-1)^3 + \lambda \|w\|_1$$

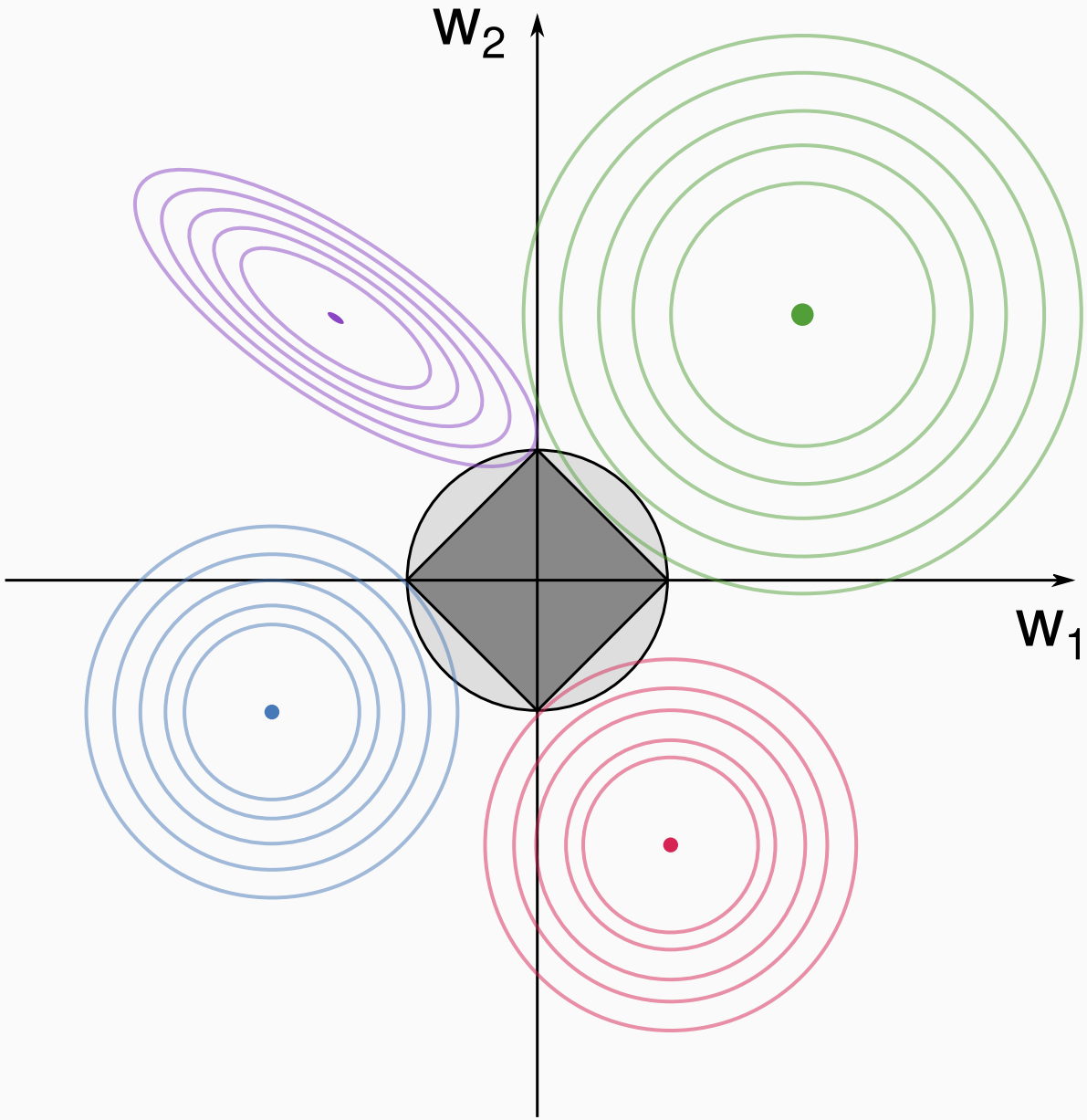


Increasing λ moves the optimum closer to 0 and away from the optimum for the loss alone.

$$E(w) = (w-1)^2 + \frac{1}{6}(w-1)^3 + \lambda \|w\|_1$$



ℓ_1 and ℓ_2 penalties with different cost functions



In general, ℓ_1 and ℓ_2 regularization are often useful when dealing with complex models and data at small scales. While they have less impact for large-scale deep learning, they still provide generalization improvements.

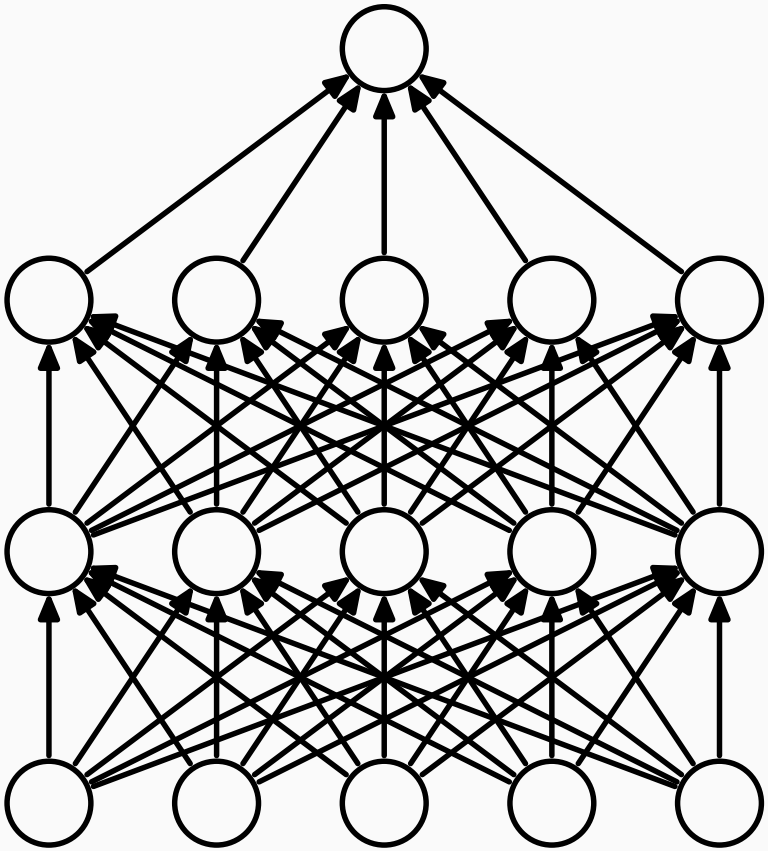
Dropout

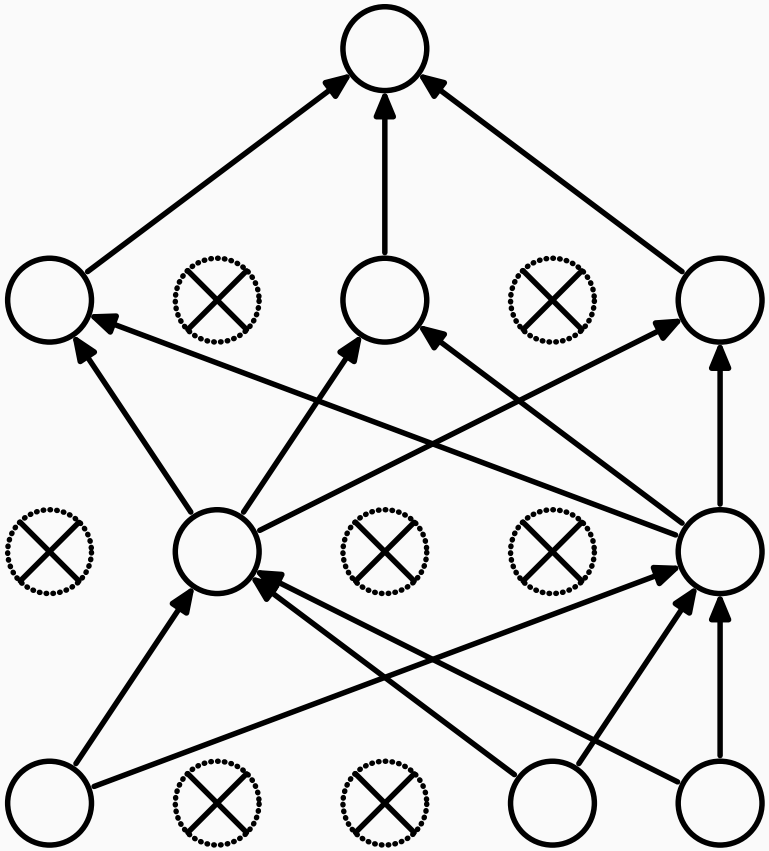
Regularization imposes restrictions on the optimization problem, aiming to achieve simpler models and avoid overfitting.

The previous regularization schemes are useful both in deep learning and other optimization problems. Deep learning has also a specific technique for reducing the complexity of the learned functions.

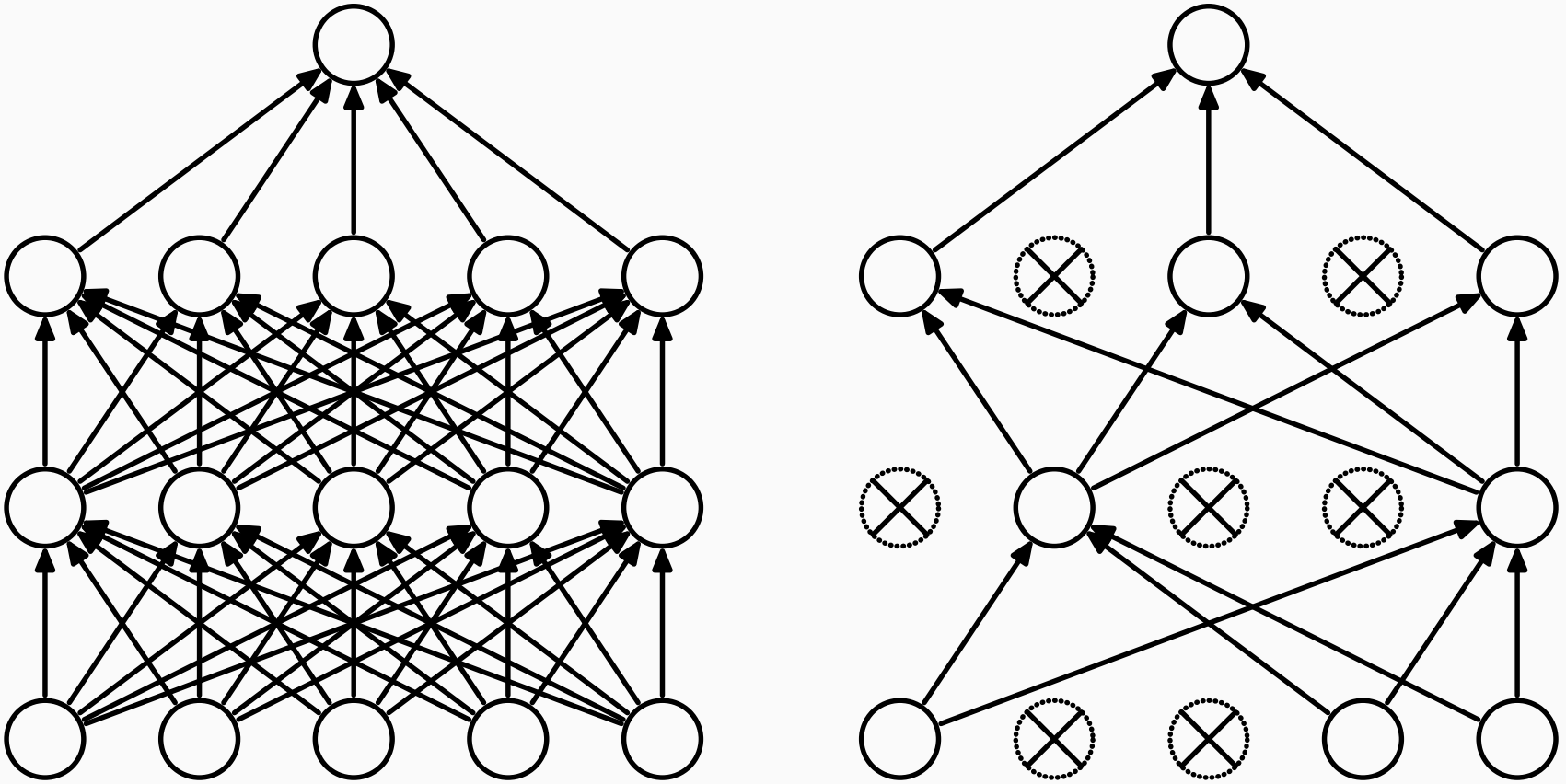
Consider the following neural networks:

Consider the following neural networks:





Consider the following neural networks:



Clearly, the model on the right is simpler with less parameters. It can be obtained by leaving out (dropping) certain neurons from the model on the left. This is the idea behind *dropout*.

How to choose which neurons to drop out?

Dropout avoids answering this question directly by *randomly* dropping out neurons each time.

When applying dropout to a layer, each neuron is retained with probability p. If a neuron is not retained, its incoming and outgoing connections are ignored, effectively removing it from the neural network.

Recall the forward pass without dropout:

$$s^{(\ell)} = w^{(\ell)} x^{(\ell-1)} + b^{(\ell)}$$
$$x^{(\ell)} = \sigma(s^{(\ell)})$$

With dropout (during training), it becomes:

$$r^{(\ell-1)} \sim \text{Bernoulli}(p)$$
$$\tilde{x}^{(\ell-1)} = x^{(\ell-1)} r^{(\ell-1)}$$
$$s^{(\ell)} = w^{(\ell)} \tilde{x}^{(\ell-1)} + b^{(\ell)}$$
$$x^{(\ell)} = \sigma(s^{(\ell)})$$

The stochasticity introduced by dropout means we cannot use the same algorithm for the forward pass in training and testing. This would mean our test performance would change every time.

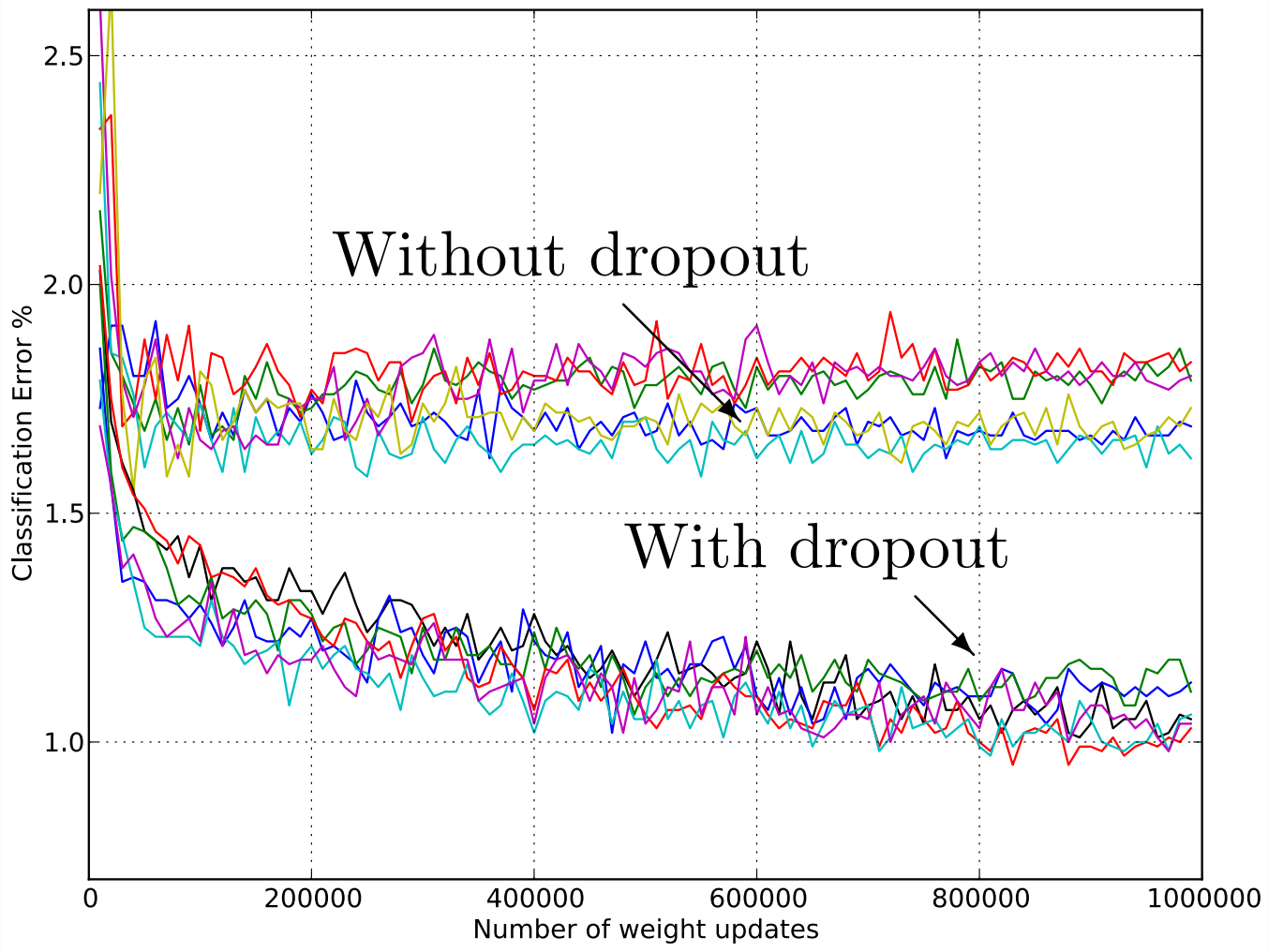
During test time, every neuron is preserved and we instead multiply the weights by p, effectively shrinking them.

One side effect of dropout is that we are dealing with a different model each time different neurons get dropped. At first, this stochasticity seems undesirable.

A common technique to quickly obtain better generalization performance with neural networks, at the cost of additional computations, is to combine multiple models with different architectures by averaging their outputs and training the whole *ensemble*.

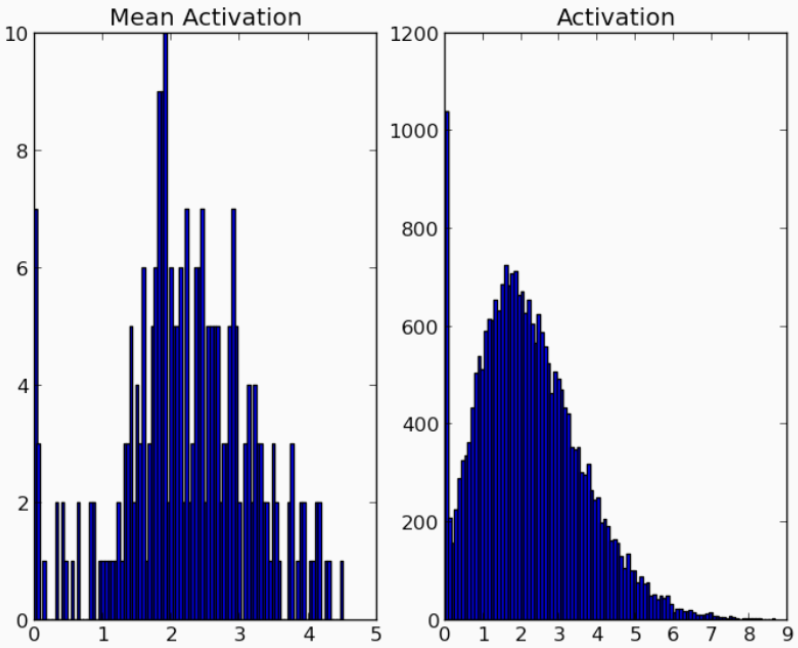
Due to how it operates, dropout essentially approximates a combination of an exponential number of different neural network architectures in an efficient procedure.

Classification error (MNIST) with multiple architectures

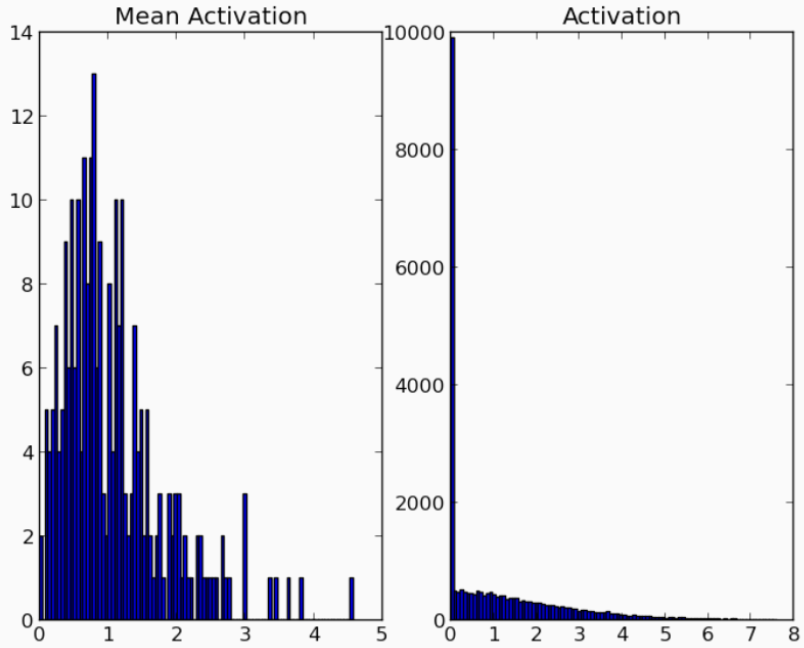


- Convergence takes longer with dropout

Dropout induces sparsity

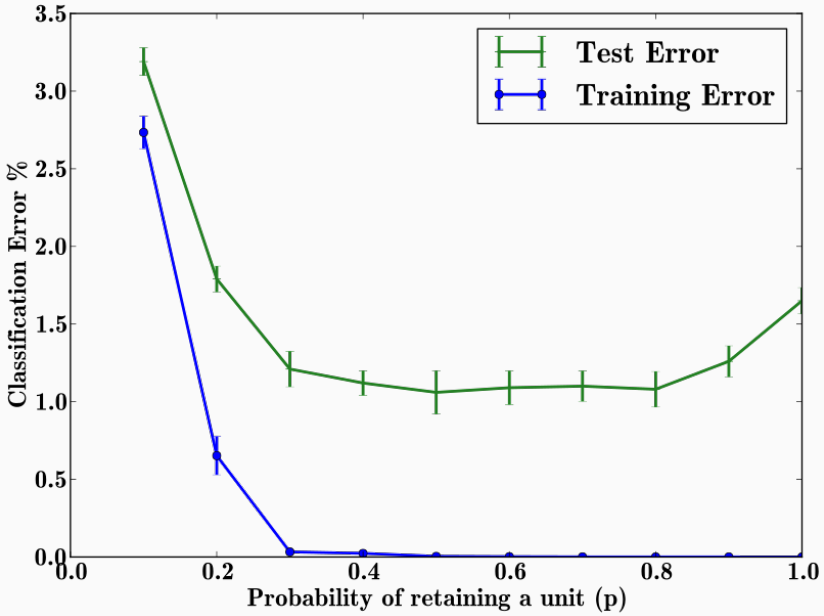


(a) Without dropout



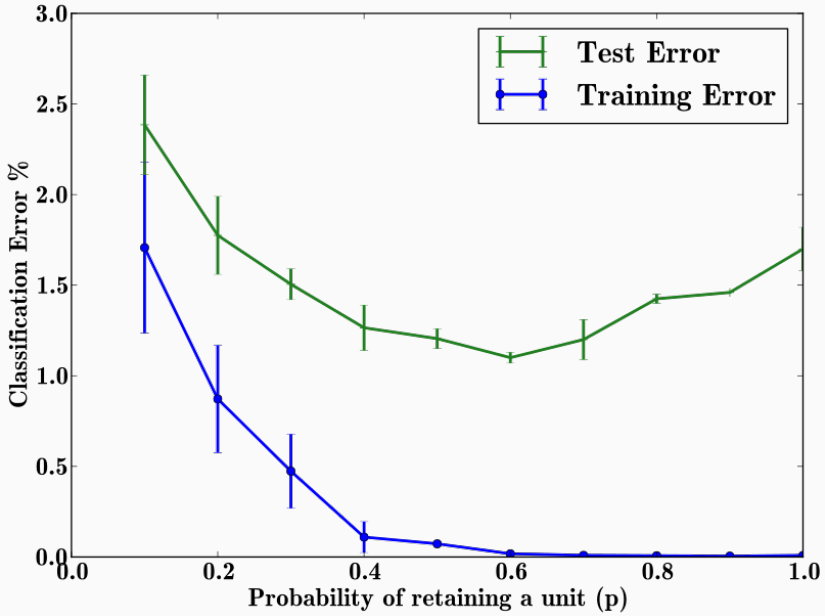
(b) Dropout with $p = 0.5$

Is there an optimal value of p?



(a) Architecture is fixed

784-2048-2048-2048-10 neurons

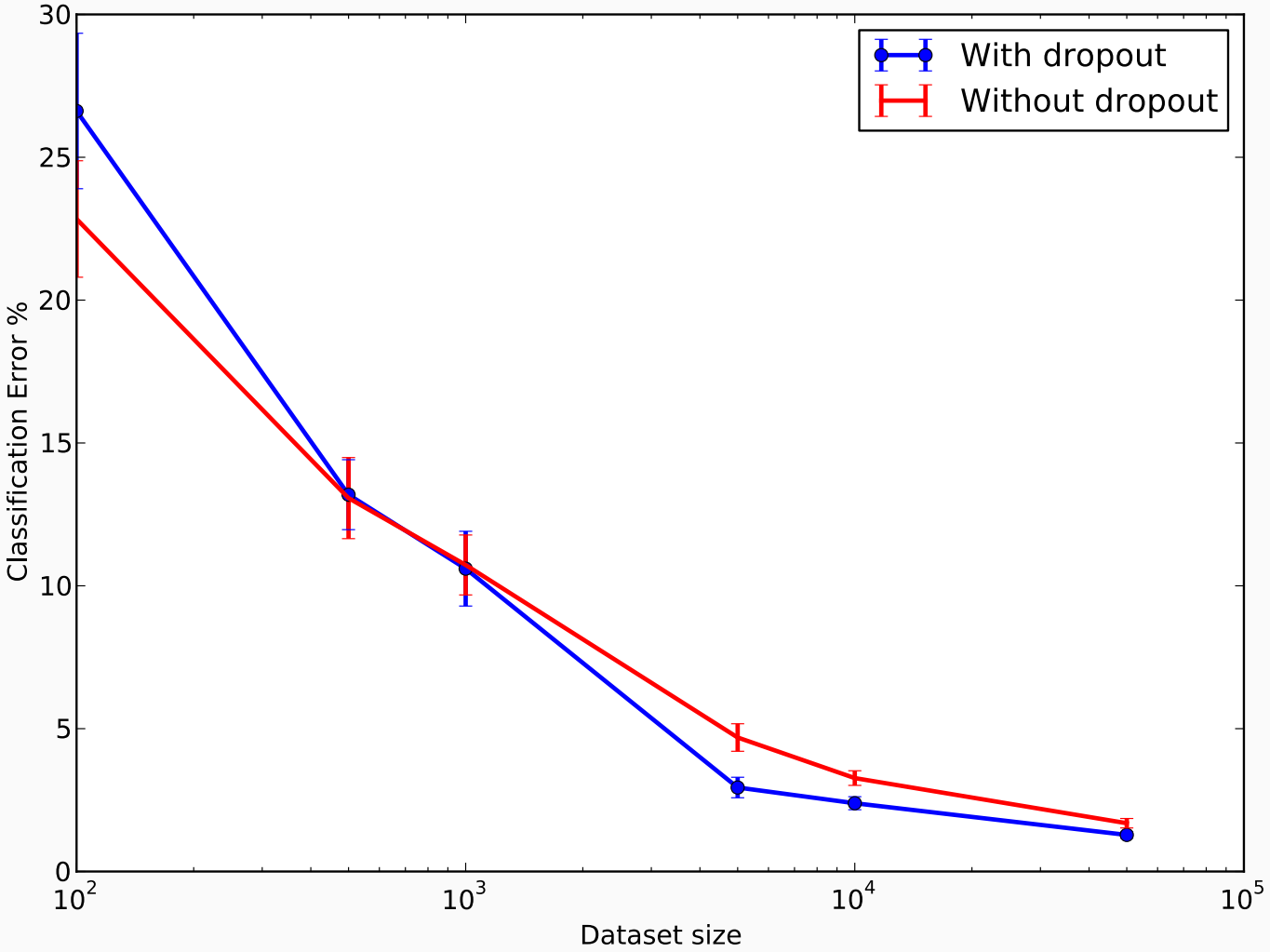


(b) Neurons $\times\, p$ is fixed

256 for first two, 512 for last hidden layer

- Range depends on each problem
- Too much dropout (low p) is worse than not having at all

Effectiveness with data size

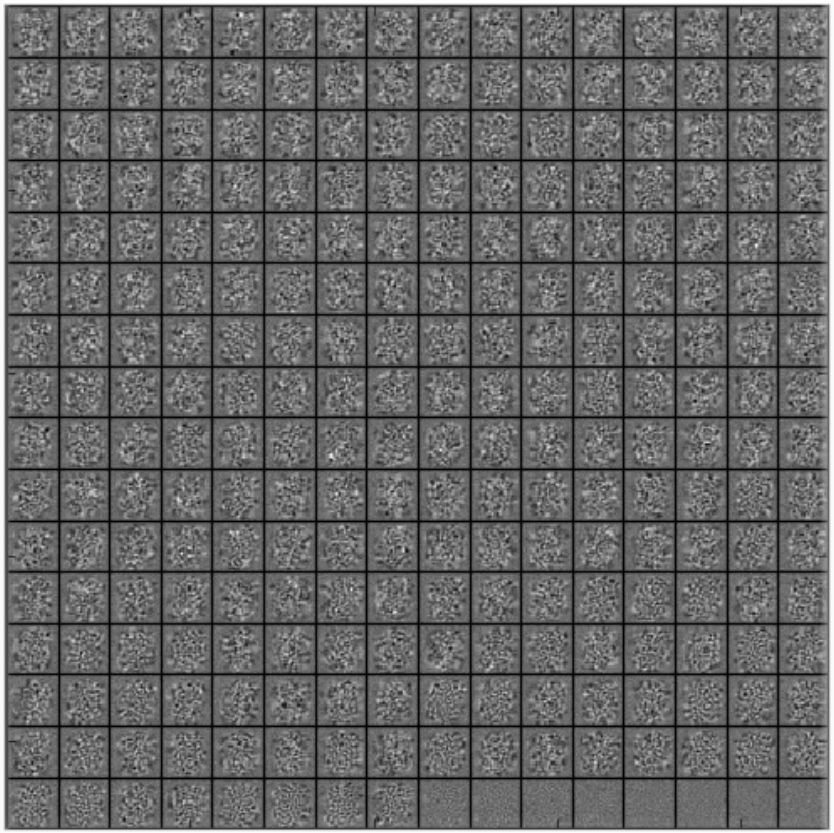


$p = 0.5$ for hidden layers, $p = 0.8$ for first layer

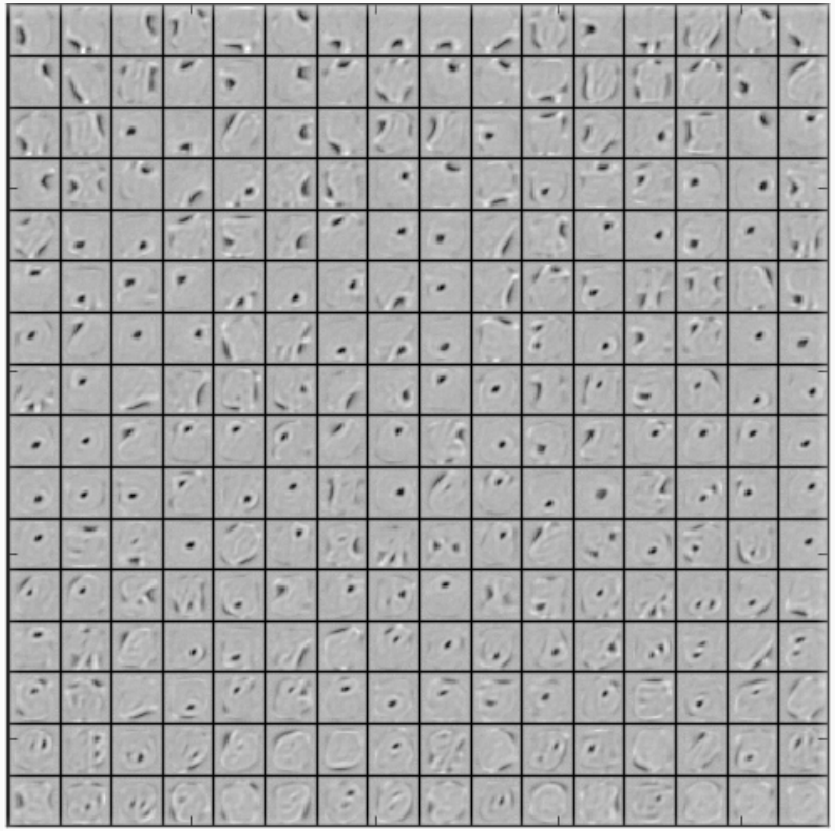
In a neural network, gradient updates are calculated for each parameter to reduce the loss given what other neurons are computing. This might result in a scenario where neurons change in a way that attempts to fix the mistakes of other neurons. This is referred to as *co-adaptation*.

Dropout fixes this problem, since it makes the presence of certain neurons (or parts of the neural network) unreliable.

Features learned (256 dimensions) on MNIST for two identical architectures with similar performance



(a) Without dropout



(b) Dropout with $p = 0.5$

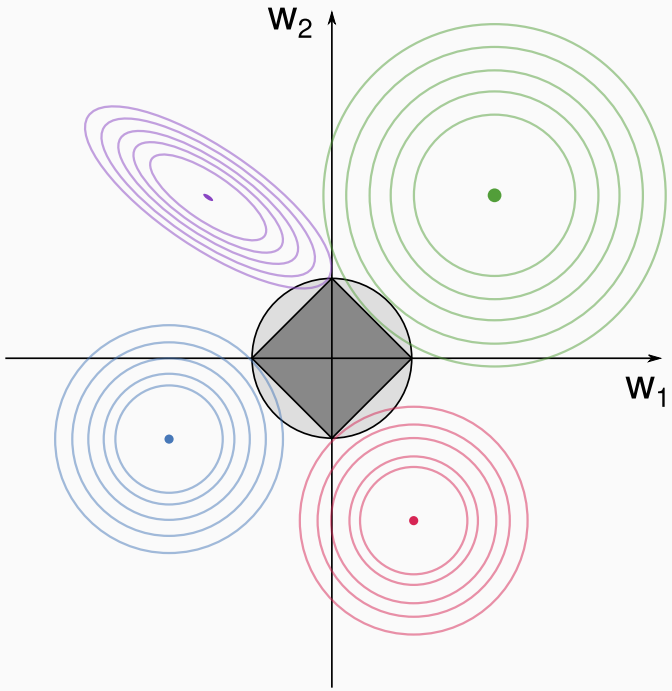
In practice, dropout is most effective on fully-connected layers, as those usually carry a much larger number of parameters. It is usual practice to employ dropout at each fully-connected layer and leave convolutional layers untouched.

There is an alternative technique for convolutional layers where, instead of dropping out pixels, we drop out entire channels. This is referred to as *spatial dropout*.

Regularization and Overfitting

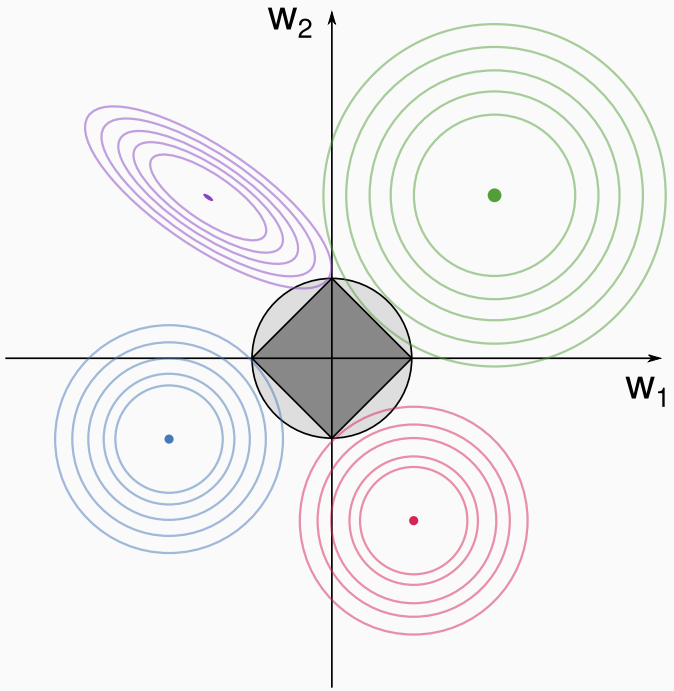
Regularization restricts the solutions of an optimization problem.

Regularization restricts the solutions of an optimization problem.

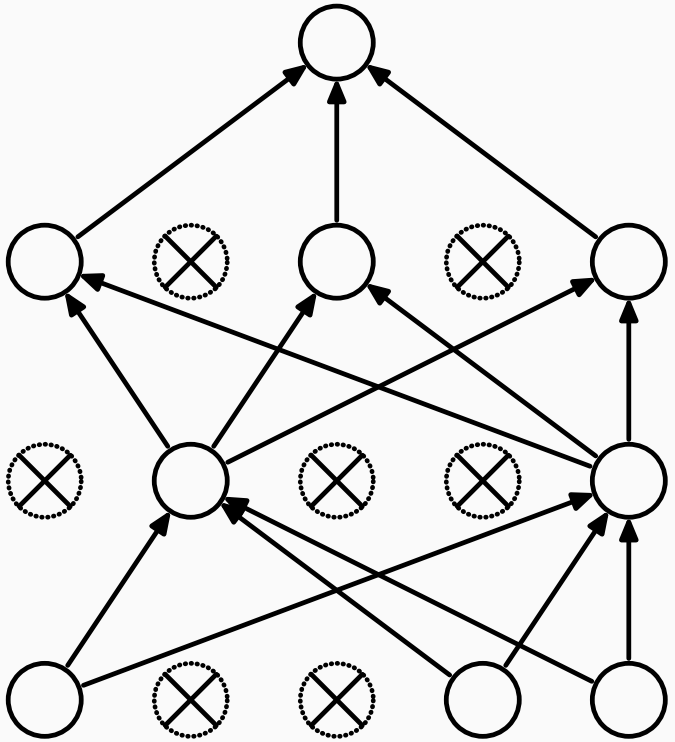


ℓ_1- & ℓ_2-regularization: restrict weight norms

Regularization restricts the solutions of an optimization problem.



ℓ_1- & ℓ_2-regularization: restrict weight norms



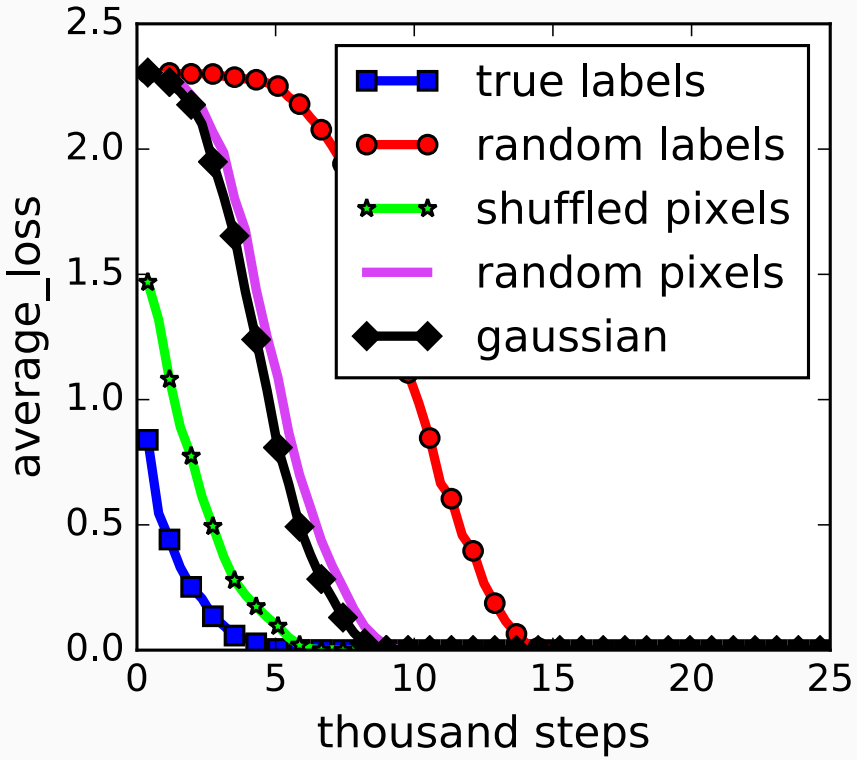
Dropout: implicitly simplifies architecture

Deep neural networks are *too* powerful. A simple MLP trained on MNIST with 2 hidden layers of 8192 neurons has already over 65 million parameters.

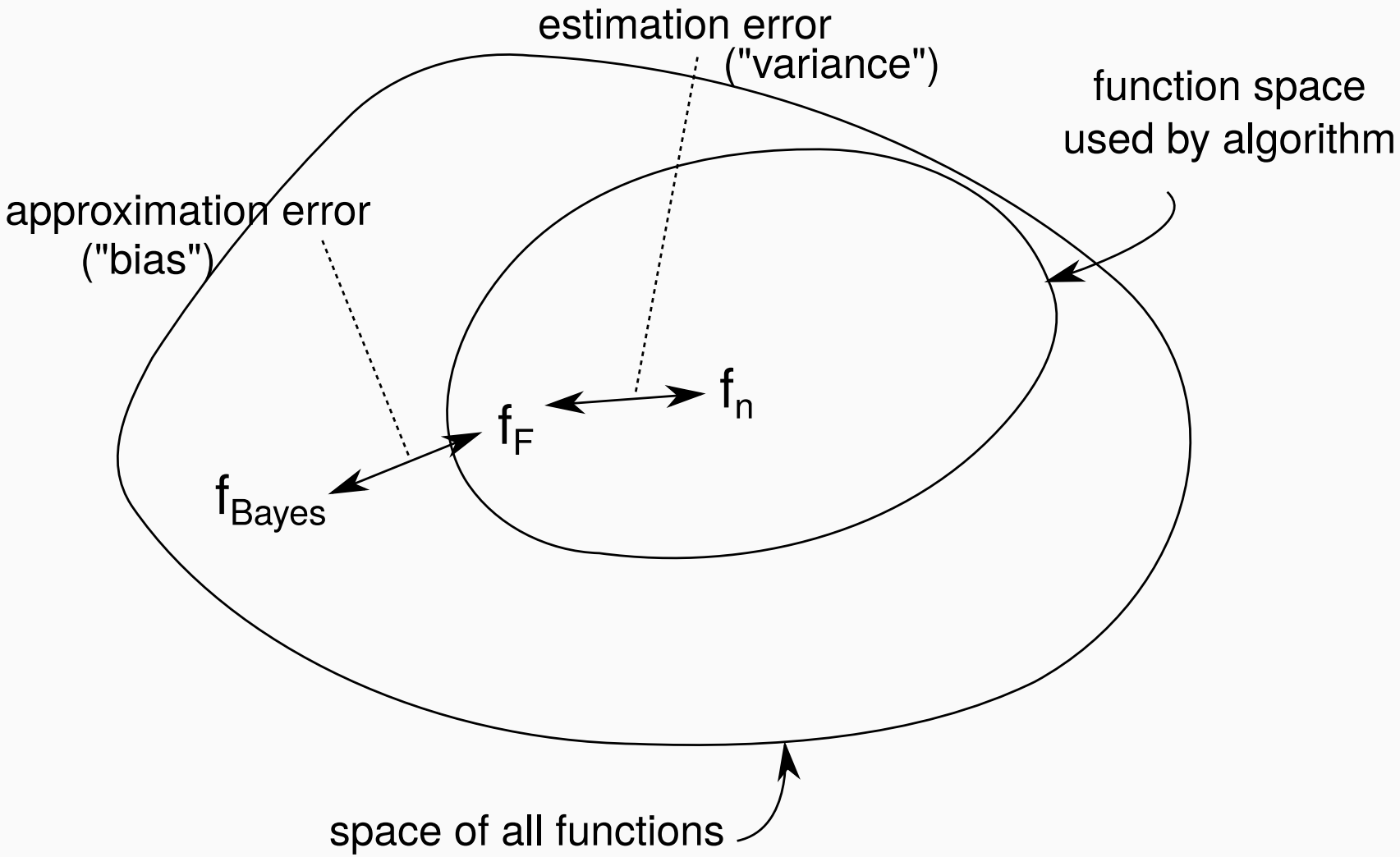
In fact, Zhang et al. (2017) show that even on completely corrupted labels or pixels, neural networks can be trained to achieve 0 training error.

Deep neural networks are *too* powerful. A simple MLP trained on MNIST with 2 hidden layers of 8192 neurons has already over 65 million parameters.

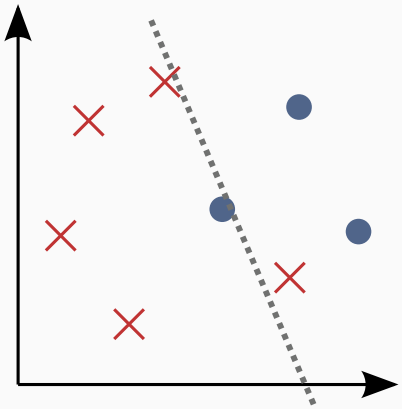
In fact, Zhang et al. (2017) show that even on completely corrupted labels or pixels, neural networks can be trained to achieve 0 training error.



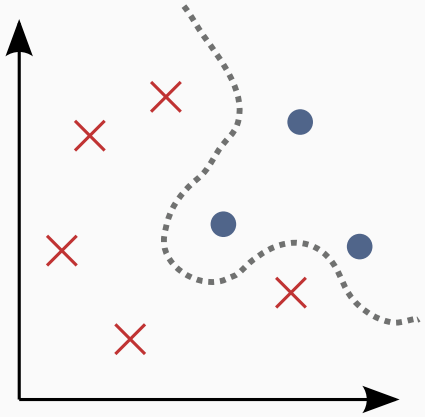
Recall the bias-variance trade-off



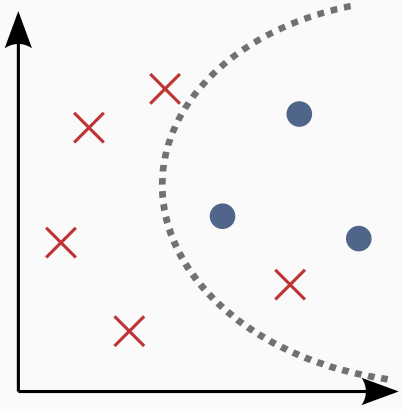
Regularization is an additional tool for controlling this trade-off



underfitting
(high bias)



overfitting
(high variance)



good
compromise

Normalizing Activations

Learning systems whose input distribution changes are said to experience *covariate shift*. Naturally, as most models assume a fixed distribution, this is detrimental to generalization performance.

Due to their nature, neural networks can be broken down into smaller models which use as inputs the outputs of the previous layers. Therefore, as training happens, they also experience covariate shift when parameters are updated. This is referred to as *internal covariate shift*.

Batch normalization, or *BatchNorm*, is a method for reducing the internal covariate shift, which improves convergence speed and reduces the chance that gradients vanish with deeper models.

This is achieved by normalizing the mean and variance of inputs, per layer, in an attempt to keep the input distribution of the next layer as fixed as possible:

$$x_i' = \beta + \gamma \frac{x_i - \mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}},$$

where μ_{batch} and σ^2_{batch} are computed per mini-batch, γ and β are learnable parameter vectors (one parameter per dimension/channel)

References

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.

C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals (2016). Understanding deep learning requires rethinking generalization. International Conference on Learning Representations.

Ioffe, S. and Szegedy, C.. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Proceedings of the 32nd International Conference on Machine Learning, in Proceedings of Machine Learning Research 37:448-456.