Scientific Machine Learning Workshop

Lecture 2: Deep Neural Networks

Ulisses Braga Neto

Department of Electrical and Computer Engineering
Scientific Machine Learning Lab (SciML Lab)
Texas A&M Institute of Data Science (TAMIDS)
Texas A&M University

Cenpes
August 2025

Neural Networks

- Neural networks (NNs) combine linear functions and univariate *nonlinearities* to produce complex discriminants with arbitrary approximation capabilities.
- A neural network consists of units called *neurons*, which are organized in *layers*.
- The neural network *architecture* specifies how the neurons are connected.
- *Deep neural networks* have complex architectures, featuring a large number of layers. These are the algorithms driving the modern revolution in artificial intelligence.

Fully-Connected Neural Networks

- The most basic neural network architecture is *fully-connected*, meaning that all neurons of a layer are connected to all neurons of the next.

- A fully-connected neural network with d inputs, L layers, and m outputs, with nonlinearity $\sigma : R \to R$, and N_ℓ neurons per layer, computes layer *activations* f_k^ℓ and layer *outputs* g_j^ℓ recursively:

$$f_k^\ell(\boldsymbol{x}) = w_{0k}^\ell + \sum_{j=1}^{N_{\ell-1}} w_{jk}^\ell \, g_j^{\ell-1}(\boldsymbol{x}), \quad \ell = 1, 2, \ldots, L;\ k = 1, \ldots, N_\ell,$$

$$g_j^\ell(\boldsymbol{x}) = \begin{cases} \sigma(f_j^\ell(\boldsymbol{x})), & \ell = 1, 2, \ldots, L-1;\ j = 1, \ldots, N_\ell, \\ x_j, & \ell = 0;\ j = 1, \ldots, d, \end{cases} \tag{1}$$

where $\boldsymbol{x} \in R^d$ is the input, $N_0 = d$, and $N_L = m$.

Fully-Connected Neural Networks

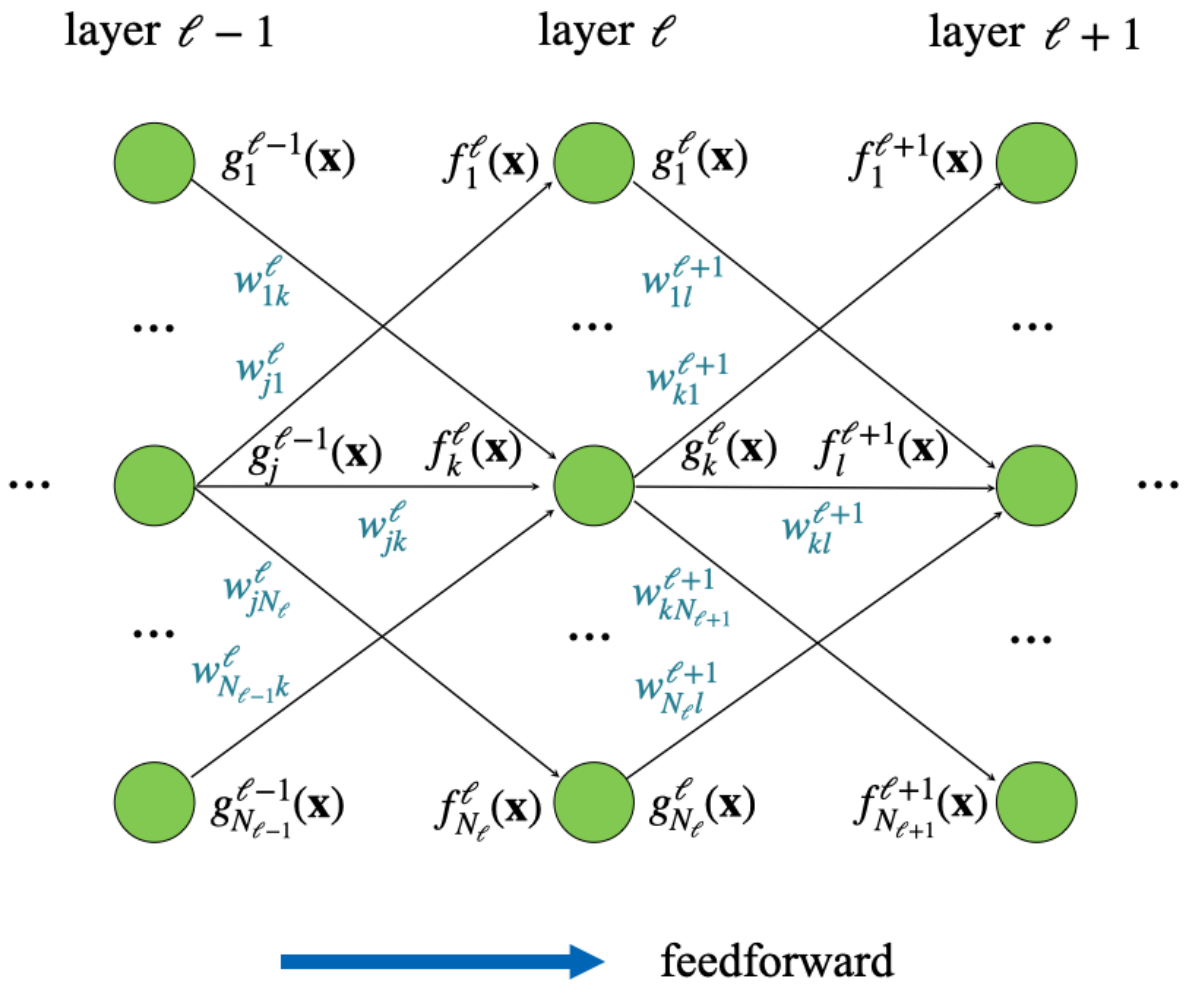

Figure: Three consecutive layers of a fully-connected neural network. (Bias weights are not represented)

Fully-Connected Neural Networks

- Layers $1, \ldots, L-1$ are the *hidden layers*, while layer L is the *output layer*.
- The smallest value $L = 2$ yields a two-layer or *shallow* network.
- The "outputs" of layer $\ell = 0$ are just the inputs x_j (sometimes, this layer is called an identity layer).
- The "activations" of layer $\ell = L$ are the neural network outputs. The single-ouput case $N_L = 1$ models a scalar function and is common in SciML.
- The *weight vector*

$$\boldsymbol{w} = [\, w_{jk}^{\ell} ; \;\; \ell = 1, 2, \ldots, L, \; j = 0, \ldots, N_{\ell-1}, \; k = 1, \ldots, N_{\ell} \,]$$

contains the parameters of the neural network. The subset of weights with $j = 0$ are called *bias weights*.

Neural Network Nonlinearities

- The nonlinearity $\sigma : R \to R$ can be of two types: saturating (or "squashing") and nonsaturating.
- The former are *sigmoids*, i.e., nondecreasing functions with bounded limits at $\pm\infty$, while the latter are unbounded.
- Common saturating nonlinearities include:
  - Step nonlinearity:

$$\sigma(x) = \begin{cases} 1, & \text{if } x > 0\,, \\ 0, & \text{if } x \leq 0\,. \end{cases}$$

  - Logistic nonlinearity:

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

  - Hyperbolic tangent ("tanh") nonlinearity:

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

Neural Network Nonlinearities

- Common nonsaturating nonlinearities include:
  - Rectifier linear unit (ReLU) nonlinearity:

$$\sigma(x) = \max(x, 0) = \begin{cases} x, & \text{if } x > 0\,, \\ 0, & \text{if } x \leq 0\,. \end{cases}$$

  - Exponential linear unit (ELU) nonlinearity:

$$\sigma(x) = \begin{cases} x, & \text{if } x > 0\,, \\ \alpha(e^x - 1), & \text{if } x \leq 0 \ (\alpha > 0)\,. \end{cases}$$

  - Swish nonlinearity:

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

- The nonlinearity determines the smoothness and expressivity of the neural network. For example, continuous nonlinearities produce continuous outputs, while the output of a ReLU neural network is a continuous piecewise linear function.

Neural Network Nonlinearities

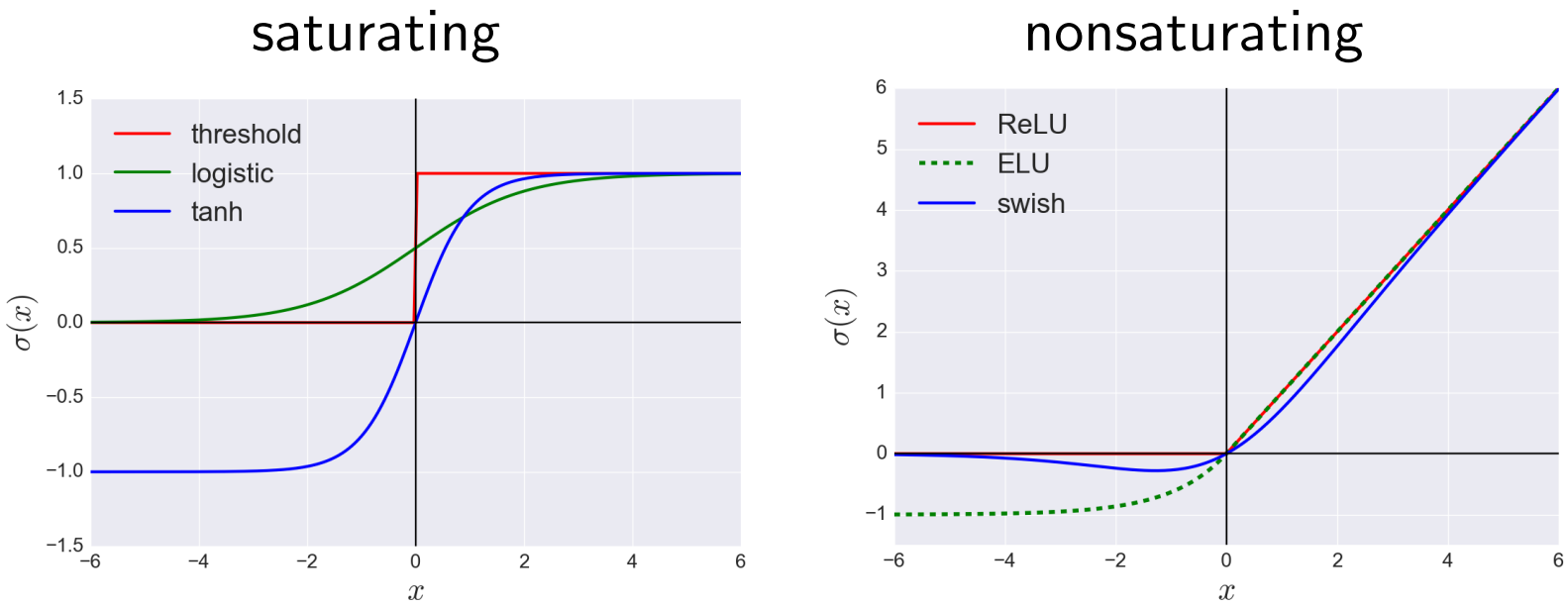

Figure: Common nonlinearities used in neural networks. (plot generated by `c06_nonlin.py`).

Universal Approximation Property of Neural Networks

- Neural networks, even shallow ones, are universal approximators. The following is the most well-known theorem that shows this.

Theorem

(Cybenko's Theorem.) Let f be a continuous function defined on the hypercube $\{0,1\}^d$ and let σ be a non-polynomial *nonlinearity. For any $\tau > 0$, there is a shallow neural network*

$$f(\boldsymbol{x};\boldsymbol{w}) = \sum_{i=0}^{k} c_i \sigma \left(\sum_{j=0}^{d} a_{ij} x_j \right) ,$$

for a choice of weights $\boldsymbol{w} = \{c_i, a_{ij}\}$, such that

$$|f(\boldsymbol{x};\boldsymbol{w}) - f(\boldsymbol{x})| < \tau , \quad \text{for all } \boldsymbol{x} \in I^d .$$

Regression

- Neural networks are used in SciML to perform *regression*, i.e. predict a numerical value Y at each point $\boldsymbol{X}$ of a physical domain.

Theorem (Data Representation)

Let $\boldsymbol{X}$, Y be jointly distributed random variables. If $E[|Y|] < \infty$, there is a function $f : R^d \to R$ and a random variable ε such that

$$Y = f(\boldsymbol{X}) + \varepsilon\,,$$

where $E[\varepsilon \mid \boldsymbol{X}] = 0$.

- This result states that the target Y can be decomposed as the sum of a function f of $\boldsymbol{X}$ and a *zero-mean additive noise* term ε.
- If ε is independent of $(\boldsymbol{X}, Y)$, the model is called *homoskedastic*, otherwise, it is said to be *heteroskedastic*.

Example: Sinusoidal Data

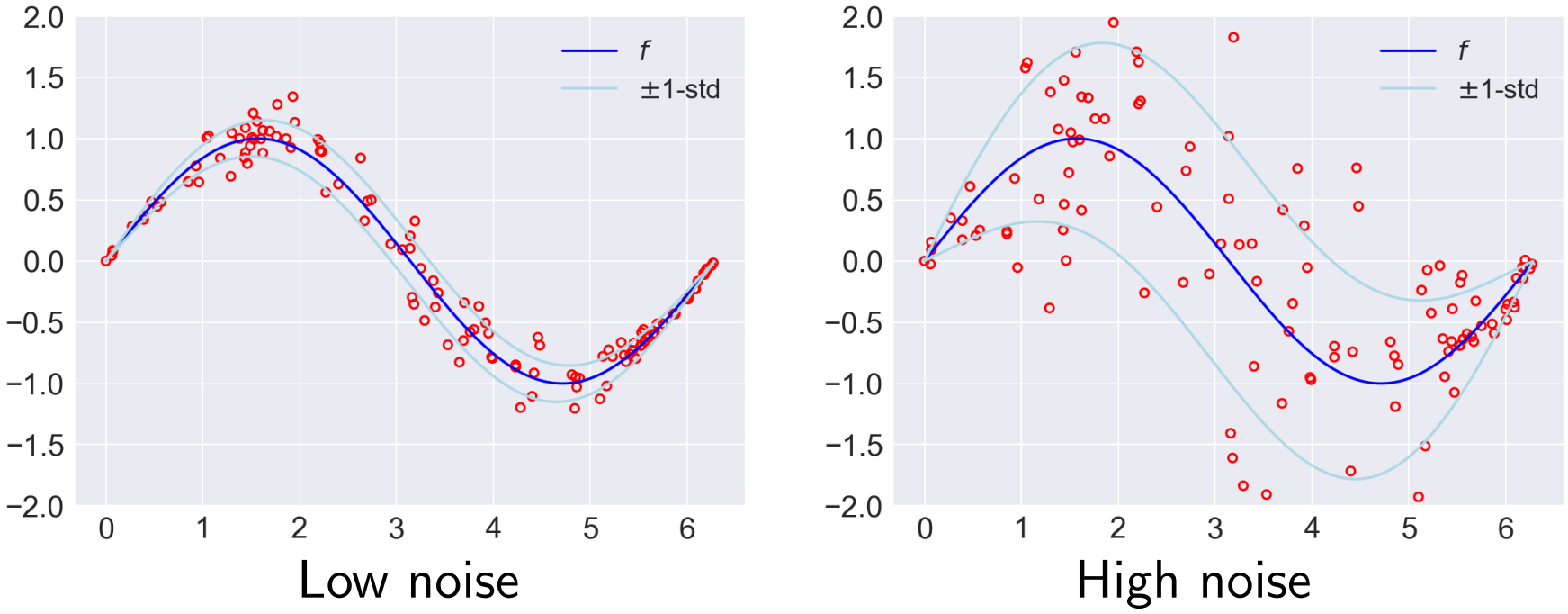

Figure: Noisy heteroskedastic sinusoidal data. Observe that f is a very good predictor of Y if the noise intensity is small, but not if it is large.

Parametric Regression

- Parametric regression exploits the Data Representation Theorem by introducing a parametric model for the unknown function f:

$$Y = f(\boldsymbol{X}; \boldsymbol{\theta}) + \varepsilon\,,$$

  where $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_m)$ is a parameter vector to be determined.

- In a *linear parametric model*, $f(\boldsymbol{x}; \boldsymbol{\theta})$ is linear in $\boldsymbol{\theta}$:

$$f(\boldsymbol{X}; \boldsymbol{\theta}) = \sum_{j=1}^{m} \theta_j \phi_j(\boldsymbol{X})\,,$$

  where ϕ_j are fixed functions, $j = 1, \ldots, m$.

- Hence, a linear parametric model performs "basis-expansion" approximation, where only the expansion coefficients θ_j are to be determined.

Least-Squares Regression

- Given sample data $S_n = \{(\boldsymbol{X}_1, Y_1), \ldots, (\boldsymbol{X}_n, Y_n)\}$, the *least-squares estimator* for parametric regression is given by

$$\hat{\boldsymbol{\theta}}_n^{\mathrm{LS}} = \arg\min_{\boldsymbol{\theta} \in \Theta} \sum_{i=1}^{n} (Y_i - f(\boldsymbol{X}_i; \boldsymbol{\theta}))^2. \tag{2}$$

- This is the maximum-likelihood estimator if ε is Gaussian.

- The minimization in (3) has a unique closed-form minimum:

$$\hat{\boldsymbol{\theta}}_n^{\mathrm{LS}} = (H^T H)^{-1} H^T \mathbf{Y}$$

where

$$H = \begin{bmatrix} \phi_1(\boldsymbol{X}_1) & \cdots & \phi_m(\boldsymbol{X}_1) \\ \vdots & \ddots & \vdots \\ \phi_1(\boldsymbol{X}_n) & \cdots & \phi_m(\boldsymbol{X}_n) \end{bmatrix}.$$

and $\mathbf{Y} = [Y_1, \ldots, Y_n]^T$ is the response vector.

Neural Network Regression

- Neural network regression (assuming the scalar output case, $N_L = 1$, and ignoring bias units for simplicity) is a form of *nonlinear* parametric regression, where

$$f(\boldsymbol{X}; \boldsymbol{w}) = \sum_{j=1}^{N_{\ell-1}} w_j^{\ell} \, g_j^{\ell-1}(\boldsymbol{x}; \boldsymbol{w} \setminus \{w_j^{\ell}\}) \, .$$

- This is still in fact a basis decomposition, but the basis functions are now not fixed, but depend, in a nonlinear fashion, on the weights (parameter) other than the output weights.

- Hence, neural networks perform adaptive-basis function approximation.

- Note that, if the weights $\boldsymbol{w} \setminus \{w_j^{\ell}\}$ are held fixed, then neural network regression *is* a linear parametric model.

Neural Network Training

- Neural network training (for regression) is a least-squares method, i.e. training means finding the optimal weight

$$\boldsymbol{w}^\star = \arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} (Y_i - f(\boldsymbol{X}_i; \boldsymbol{w}))^2. \tag{3}$$

- In the case of neural network regression, the least-squares criterion $\mathcal{L}(\boldsymbol{w}) = \sum_{i=1}^{n} (Y_i - f(\boldsymbol{X}_i; \boldsymbol{w}))^2$ is called the *loss*.
- Since neural network regression is nonlinear, there is unfortunately no unique, closed-form solution.
- Instead, training proceeds by adjusting the weights $\boldsymbol{w}$ to find a (good) local inimum of the *loss surface* $\mathcal{L}(\boldsymbol{w})$.

Example: Neural Network Regression with ReLU

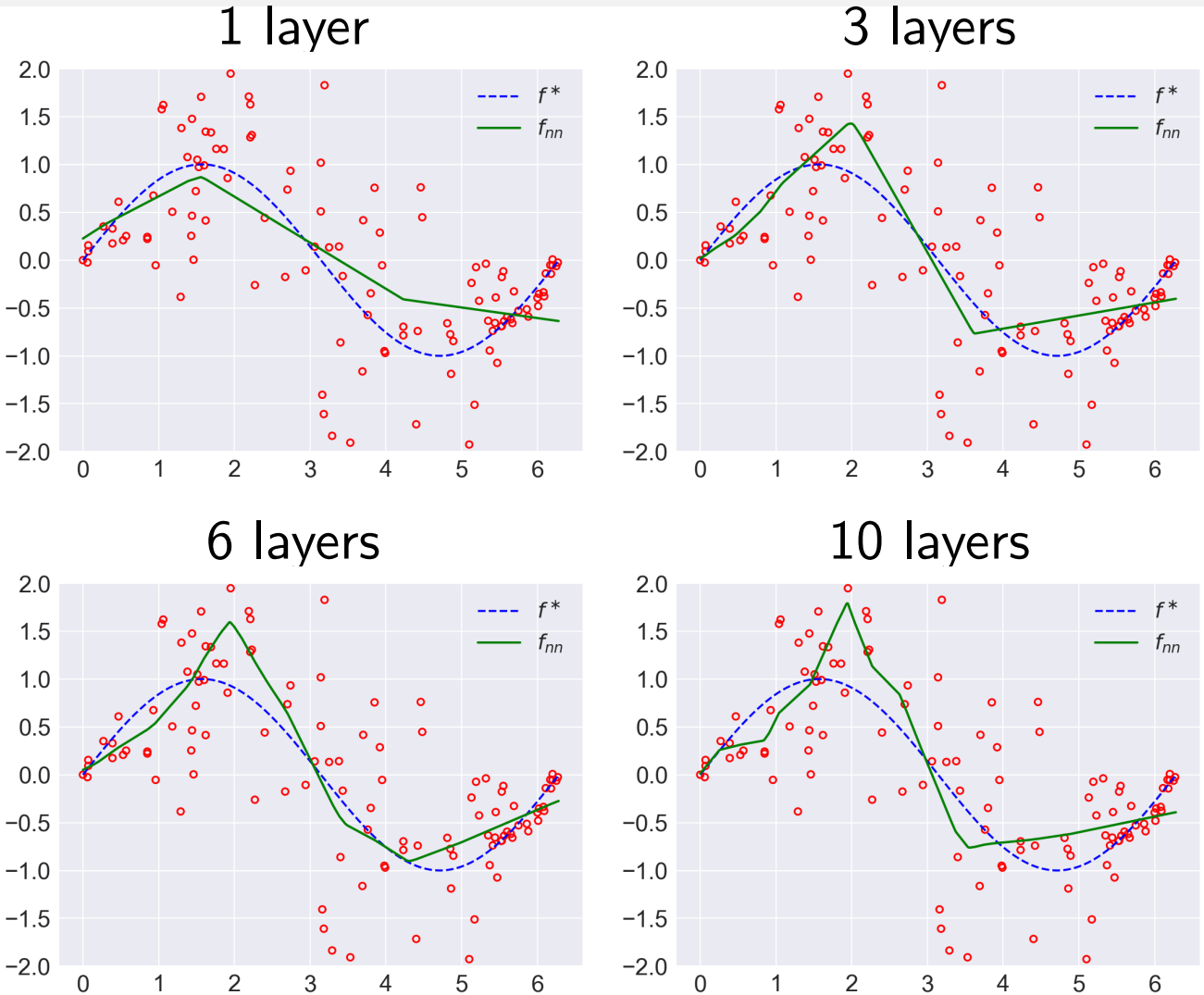

Figure: The output is piecewise linear. Underfitting and overfitting are visible as the number of layers varies. (Plots generated by `c11_nnet.py`.)

Example: Neural Network Regression with tanh

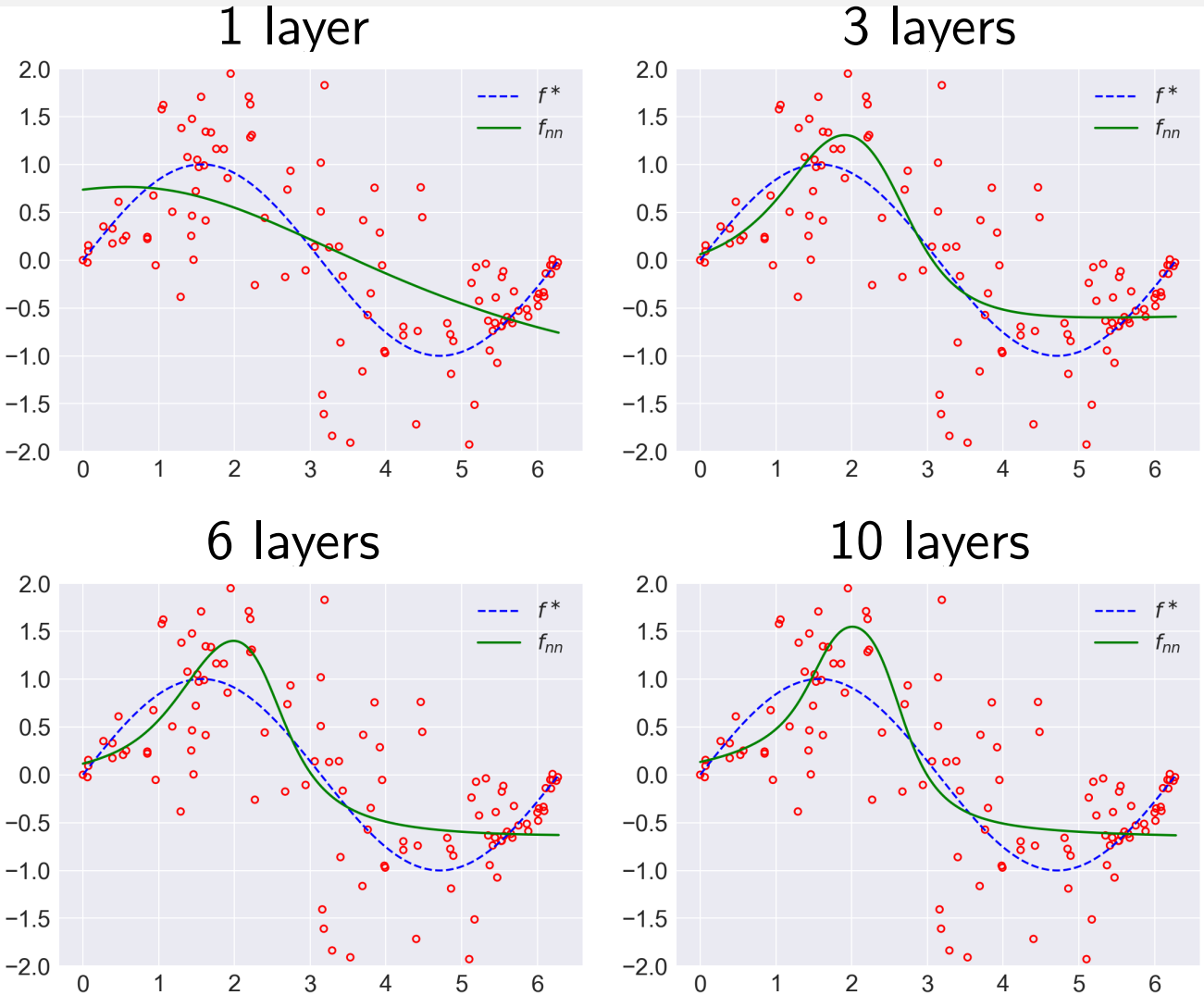

Figure: The output is infinitely differentiable. Underfitting and overfitting are visible at different layer numbers. (Plots generated by `c11_nnet.py`.)

Backpropagation Algorithm

- We recall that training a neural network consists in solving the following *unconstrained minimization problem*:

$$\min \mathcal{L}(\boldsymbol{w})$$
$$\boldsymbol{w} \in R^p$$

  where p is the number of weights.

- This is a highly nonconvex, high-dimensional problem, and we can only hope to find a good local minimum $\boldsymbol{w}^*$.

- The most common approach to do this is via gradient descent, where one starts with a randomly initialized weight vector $\boldsymbol{w} = \boldsymbol{w}_0$, and performs the recursion

$$\boldsymbol{w}^{k+1} = \boldsymbol{w}^k + \eta_k \boldsymbol{v}^k, \quad k = 0, 1, \ldots \tag{4}$$

  until a suitable termination criterion is achieved, where $\eta_k > 0$ is the *learning rate* and $\boldsymbol{v}^k$ is a direction vector.

Backpropagation Algorithm

- Termination occurs after a prescribed number of iterations or after the loss function has not decreased after a while.
- Termination may also occur after a computational budget (e.g. number of iterations) has been exhausted.
- It is easy to show that for a greedy step, i.e., to go in the direction of maximum decrease in $\mathcal{L}$, one should set $\boldsymbol{v}^k = -\nabla\mathcal{L}(\boldsymbol{w}^k)$, where

$$\nabla\mathcal{L}(\boldsymbol{w}) = \left[\frac{\partial\mathcal{L}(\boldsymbol{w})}{\partial w_{jk}^{\ell}};\ \ell = 1, 2, \ldots, L,\ j = 0, \ldots, N_{\ell-1},\ k = 1, \ldots, N_{\ell}\right]$$

  is the gradient of the loss function.

Backpropagation Algorithm

- The partial derivatives in the previous equation can be computed efficiently through a classical procedure known as the *backpropagation algorithm*, described next.

- First, note that the loss functions discussed in the previous section are of the form

$$\mathcal{L}(\boldsymbol{w}) = \sum_{i=1}^{n} \mathcal{L}_i(\boldsymbol{w}),$$

  where $\mathcal{L}_i(\boldsymbol{w})$ is the loss for a single training point $(\boldsymbol{x}_i, y_i)$.

- Hence, the partial derivatives are given by the sum of the contributions of all training points (omitting the dependence on the weights for simplicity):

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{\ell}} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}_i}{\partial w_{jk}^{\ell}}. \tag{5}$$

Backpropagation Algorithm

- Using the chain rule of differentiation and (1), we have

$$\frac{\partial \mathcal{L}_i}{\partial w_{jk}^{\ell}} = \frac{\partial \mathcal{L}_i}{\partial f_k^{\ell}} \frac{\partial f_k^{\ell}}{\partial w_{jk}^{\ell}} = \bar{f}_k^{\ell}\, g_j^{\ell-1}(\boldsymbol{x}_i)\,, \tag{6}$$

where we define $g_0^{\ell-1}(\boldsymbol{x}_i) \equiv 1$ to account for the bias weights, and the activation *adjoint* $\bar{f}_k^{\ell}$ can be computed recursively using the chain rule (see Figure 1):

$$\bar{f}_k^{\ell} = \frac{\partial \mathcal{L}_i}{\partial f_k^{\ell}} = \sum_{l=1}^{N_{\ell+1}} \frac{\partial \mathcal{L}_i}{\partial f_l^{\ell+1}} \frac{\partial f_l^{\ell+1}}{\partial f_j^{\ell}} = \sum_{l=1}^{N_{\ell+1}} \bar{f}_l^{\ell+1} \frac{\partial f_l^{\ell+1}}{\partial f_k^{\ell}}\,,$$

while

$$\frac{\partial f_l^{\ell+1}}{\partial f_k^{\ell}} = \frac{\partial f_l^{\ell+1}}{\partial g_k^{\ell}} \frac{\partial g_k^{\ell}}{\partial f_k^{\ell}} = w_{kl}^{\ell+1} \sigma'(f_k^{\ell}(\boldsymbol{x}_i))\,.$$

Backpropagation Algorithm

- This results in the *backpropagation equation*:

$$\bar{f}_k^\ell \;=\; \sigma'(f_k^\ell(\boldsymbol{x}_i)) \sum_{l=1}^{N_{\ell+1}} w_{kl}^{\ell+1}\, \bar{f}_l^{\ell+1}\,. \tag{7}$$

- Hence, the adjoints at one hidden layer are backpropagated from the adjoints at the next layer.
- In sum, the procedure for one "backprop" iteration is as follows:
  1. Given a training input $\boldsymbol{x}_i$, the current neural network, with the current weights, is run in feedforward mode to update all activations $f_k^\ell(\boldsymbol{x}_i)$ and neuron outputs $g_j^\ell(\boldsymbol{x}_i)$.
  2. Next, the adjoints for the output layer are computed. Taking as an example the mean square loss (**??**) for a network with a single output $f^L(\boldsymbol{x})$, this gives

$$\bar{f}^L \;=\; \frac{\partial \mathcal{L}_i}{\partial f^L} \;=\; \frac{\partial}{\partial f^L}\left(\frac{1}{2}(y_i - f^L(\boldsymbol{x}_i)^2)\right) \;=\; y_i - f^L(\boldsymbol{x}_i)\,.$$

  This is the error between the network prediction and the true label.

Backpropagation Algorithm

3. Then the adjoints at the previous, next-before-last layer are computed using (7):

$$\bar{f}_k^{L-1} = \sigma'(f_k^{L-1}(\boldsymbol{x}_i)) \cdot w_k^L \bar{f}^L, \quad k = 1, \ldots, N_{L-1}.$$

   This "propagates" the error at the ouput layer "back" to the next-before-last layer, using the current values of the output weights w_k^L.

4. The process is repeated to propagate the error back to all previous layers using (7). This corresponds to running the neural network in backward mode and is an example of *reverse-mode automatic differentiation* applied to the neural network computational graph.

5. Once all the adjoints are found, the partial derivatives in (6) are computed.

Backpropagation Algorithm

6. The process is repeated for all training points and the loss gradient is computed using (5).
7. Finally, the weights are updated according to (4).

- From (7), we can see that the slope $\sigma'(x)$ of the nonlinearity at the current value of the activation f_k^ℓ affects the magnitude of $\bar{f}_k^i$ and thus the magnitude of the partial derivative in (6).
- Hence, the derivative of the nonlinearity plays an important role in backprop training.
- In particular, note that the step nonlinearity has no derivative at the origin, and zero derivative elsewhere; hence, it cannot be used with gradient-based training.

Weight Initialization

- The traditional method had been to initialize all weights as independent zero-mean Gaussian random variables with unit variance.
- However, when consecutive layers feature different numbers of neurons n_{in} and n_{out}, the activations when the neural network is run in feedforward mode will have a different variance than the activation adjoints when the neural network is run in reverse mode for training.
- This was observed to create a vanishing gradient problem.

Weight Initialization

- To address this problem, "Xavier" or "Glorot" initialization (named after one of the inventors) uses a common value for the variance equal to the reciprocal of the average number of neurons:

$$\sigma^2 = \frac{1}{n_{avg}} = \frac{2}{n_{in} + n_{out}}.$$

- It was observed empirically that this produces gradients with uniform variance over all layers, in the case of symmetric nonlinearities, such as tanh.

- For asymmetric nonlinearities, such as the ReLU, ELU, and swish, it has been found that the "He" initialization (again named after one of the inventors) is better:

$$\sigma^2 = \frac{2}{n_{in}}.$$

Stochastic Gradient Descent

- The backpropagation algorithm needs to run the neural network in feedfoward and backward mode for each training data point, for a single iteration of gradient descent.
- This approach is infeasible for large data sets with thousands or millions of data, which are common in modern deep learning, including in SciML.
- In *stochastic gradient descent* (also known as mini-batch training) a random sample of a fixed number of training points (the size can be anywhere between a single point and the entire data set) is taken to compute a "noisy" gradient.
- Not only can this speed up training significantly, but there is also evidence that it allows gradient descent to escape poor local minima that prevent full-batch training from succeeding.

Learning Rate Schedule

- The loss surface of deep neural networks can be quite complex, and as a result, the sequence of learning rates $\{\eta_k; k = 1, 2, \ldots\}$ must be set carefully for successful training.
- Typically, one wants the learning rate to be large in the beginning of training, to make rapid progress, and small at the end of training, to allow training to converge.
- If the learning rate does not decrease sufficiently fast as training progresses, gradient descent will either diverge (infamous "NaNs" will appear) or bounce around a sharp minimum without being able to enter it. The latter is a particularly serious problem if stochastic gradient descent is used.
- On the other hand, if the learning rate decreases too fast, then progress is not made, and convergence will take arbitrarily long.

Learning Rate Schedule

- The *learning rate schedule* is therefore a critical component for successfully training deep neural networks. This schedule must usually be set before training starts.
- A popular approach is the exponential learning rate schedule:

$$\eta^k = \eta^0 0.1^{k/s}, \quad k = 0, 1, \ldots$$

- This makes the learning rate decrease by a factor of 10 after every s steps. The values of η_0 and s are often tuned by trial and error.
- Another approach is the piecewise constant schedule, where the learning rate is set to a different prespecified constant value over different intervals of the training step k.
- This approach provides more flexibility than the exponential schedule, at the cost of having more hyperparameters to tune.

Optimization with Momentum and Adaptive Gradients

- Even if the learning rate schedule is set carefully, gradient descent can take too large steps in regions of high curvature of the loss (creating oscillation), or too small steps in flat regions of the loss.
- We describe next a series of optimization methods, of increasing complexity, that can improve the performance of gradient descent in deep learning:
  - "Heavy-Ball" method.
  - Nesterov method.
  - AdaGrad method.
  - RMS Prop method.
  - ADAM method.
  - Second-order methods.

"Heavy-Ball" Method

- The so-called *heavy-ball* algorithm has the following update:

$$\begin{aligned} \boldsymbol{m}^k &= \beta \boldsymbol{m}^{k-1} - \eta_k \nabla \mathcal{L}(\boldsymbol{w}^k)\,, \\ \boldsymbol{w}^{k+1} &= \boldsymbol{w}^k + \boldsymbol{m}^k\,, \end{aligned}$$

  where $\boldsymbol{m}$ is the *momentum vector*, and $0 < \beta < 1$ is an exponential decay parameter.

- If $\beta = 0$ this reduces to the usual gradient descent method, while $\beta \geq 1$ is unstable.

- A value of $\beta = 0.9$ is common.

Nesterov Method

- The *Nesterov* method is a small variation of the heavy ball method, which accelerates convergence by computing the gradient not at the current value of $\boldsymbol{w}$, but at a "future" value.
- The intuition is that the gradient at a point closer to the minimum is more desirable, as it should point more toward the minimum.
- The Nesterov update is

$$\begin{aligned} \boldsymbol{m}^k &= \beta \boldsymbol{m}^{k-1} - \eta_k \nabla \mathcal{L}(\boldsymbol{w}^k + \beta \boldsymbol{m}^{k-1}) \,, \\ \boldsymbol{w}^{k+1} &= \boldsymbol{w}^k + \boldsymbol{m}^k \,, \end{aligned}$$

  where $0 < \beta < 1$ as before.
- Again, $\beta = 0.9$ is a popular choice.

AdaGrad Method

- Momentum does not deal directly with the learning rate.
- One would like that to be smaller along the directions of steepest descent, and larger in the directions where there is little gradient.
- The *AdaGrad* (adaptive gradient descent) method accomplishes this with the update

$$\begin{aligned} \boldsymbol{s}^k &= \boldsymbol{s}^{k-1} + (\nabla \mathcal{L}(\boldsymbol{w}^k))^2 \\ \boldsymbol{w}^{k+1} &= \boldsymbol{w}^k - \eta_k \frac{\nabla \mathcal{L}(\boldsymbol{w}^k)}{\sqrt{\boldsymbol{s}^k + \varepsilon}} \end{aligned}$$

where $\varepsilon > 0$ is a small number to avoid division by zero, and all vector operations are performed componentwise.

RMSProp Method

- *RMS Prop* is a popular variation of Adagrad, which introduces exponential decay, so that older gradients are "forgotten".
- The RMS Prop update is

$$\begin{aligned} \boldsymbol{s}^k &= \beta \boldsymbol{s}^{k-1} + (1-\beta)(\nabla \mathcal{L}(\boldsymbol{w}^k))^2 \\ \boldsymbol{w}^{k+1} &= \boldsymbol{w}^k - \eta_k \frac{\nabla \mathcal{L}(\boldsymbol{w}^k)}{\sqrt{\boldsymbol{s}^k + \varepsilon}} \end{aligned}$$

  where $0 < \beta < 1$ is the exponential decay parameter.
- A value of $\beta = 0.9$ is often used.

ADAM Method

- Finally, the very popular *Adam* method is a hybrid between the momentum-based and Adagrad-based classes of methods.
- The Adam update is

$$\begin{aligned} \boldsymbol{m}^k &= \beta_1 \boldsymbol{m}^{k-1} - (1-\beta_1)\,\eta_k \nabla \mathcal{L}(\boldsymbol{w}^k) \\ \boldsymbol{s}^k &= \beta_2 \boldsymbol{s}^{k-1} + (1-\beta_2)(\nabla \mathcal{L}(\boldsymbol{w}^k))^2 \\ \widehat{\boldsymbol{m}}^k &= \boldsymbol{m}^k/(1-\beta_1^k) \\ \widehat{\boldsymbol{s}}^k &= \boldsymbol{s}^k/(1-\beta_2^k) \\ \boldsymbol{w}^{k+1} &= \boldsymbol{w}^k + \eta_k \frac{\widehat{\boldsymbol{m}}^k}{\sqrt{\widehat{\boldsymbol{s}}^k + \varepsilon}} \end{aligned}$$

  where $0 < \beta_i < 1$, $i = 1, 2$.
- The popular choices are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.
- "Nadam" employs the Nesterov look-ahead technique instead.

Second-Order Optimization Methods

- Another possibility to speed up training is to use second-order derivative information.
- The multivariate Taylor-series truncated at second-order terms approximates the loss function at the current value of the parameters by a quadratic surface:

$$\mathcal{L}(\boldsymbol{w}) \approx \mathcal{L}(\boldsymbol{w}^k) + \nabla\mathcal{L}(\boldsymbol{w}^k)(\boldsymbol{w}-\boldsymbol{w}^k) + \frac{1}{2}(\boldsymbol{w}-\boldsymbol{w}^k)^T\nabla^2\mathcal{L}(\boldsymbol{w}^k)(\boldsymbol{w}-\boldsymbol{w}^k)\,,$$

  where $\nabla^2\mathcal{L}$ is the Hessian, which is a $p \times p$ matrix of second-order partial derivatives.
- This is a convex surface with a single strict minimum $\boldsymbol{w}^*$ (unless the Hessian is zero). By differentiation with respect to $\boldsymbol{w}$, the minimum $\boldsymbol{w}^*$ must satisfy $-\nabla\mathcal{L}(\boldsymbol{w}^k) + \nabla^2\mathcal{L}(\boldsymbol{w}^k)(\boldsymbol{w}^* - \boldsymbol{w}^k) = 0$, i.e. $\boldsymbol{w}^* = \boldsymbol{w}^k - (\nabla^2\mathcal{L}(\boldsymbol{w}^k))^{-1}\nabla\mathcal{L}(\boldsymbol{w}^k)$.

Second-Order Optimization Methods

- Accordingly, the *Newton method* update is

$$\boldsymbol{w}^{k+1} = \boldsymbol{w}^k - (\nabla^2 \mathcal{L}(\boldsymbol{w}^k))^{-1} \nabla \mathcal{L}(\boldsymbol{w}^k)\,.$$

- This method gets its name from the classical Newton method for root-finding: the minimization of a quadratic function $f(\boldsymbol{x})$ is equivalent to finding the root of $\nabla f(\boldsymbol{x}) = 0$.

- The pure Newton method is impractical for deep neural network training because it requires storing and inverting a $p \times p$ Hessian matrix, where p is the number of weights, which can be very large (see Exercise 6.5).

- *Quasi-Newton* methods avoid computing and inverting the Hessian.