Operations Research
10. Neural Network (NN)

MLP / NN Architecture

- The objective: Using an **MultiLayer Perceptron** (MLP) / **Neural Network** (NN), denoted as $\mathcal{F}$, to approximate a function $\mathbf{f} : \mathbf{x} \in \mathbb{R}^d \rightarrow \mathbf{y} \in \mathbb{R}^D$
- Computing units of an MLP / NN, called **artificial neurons**, are stacked in a number of consecutive layers
- The zeroth layer of $\mathcal{F}$ is called the **source layer**, which is not a computing layer but is only responsible for providing an input (of dimension d) to the network
- The last layer of $\mathcal{F}$ is called the **output layer**, which outputs the network's prediction of dimension D
- Every other layer in between is called a **hidden layer**; the number of neurons in a layer defines the **width** of that layer

A schematic of an MLP / NN with 2 hidden layers is shown in Figure 1.

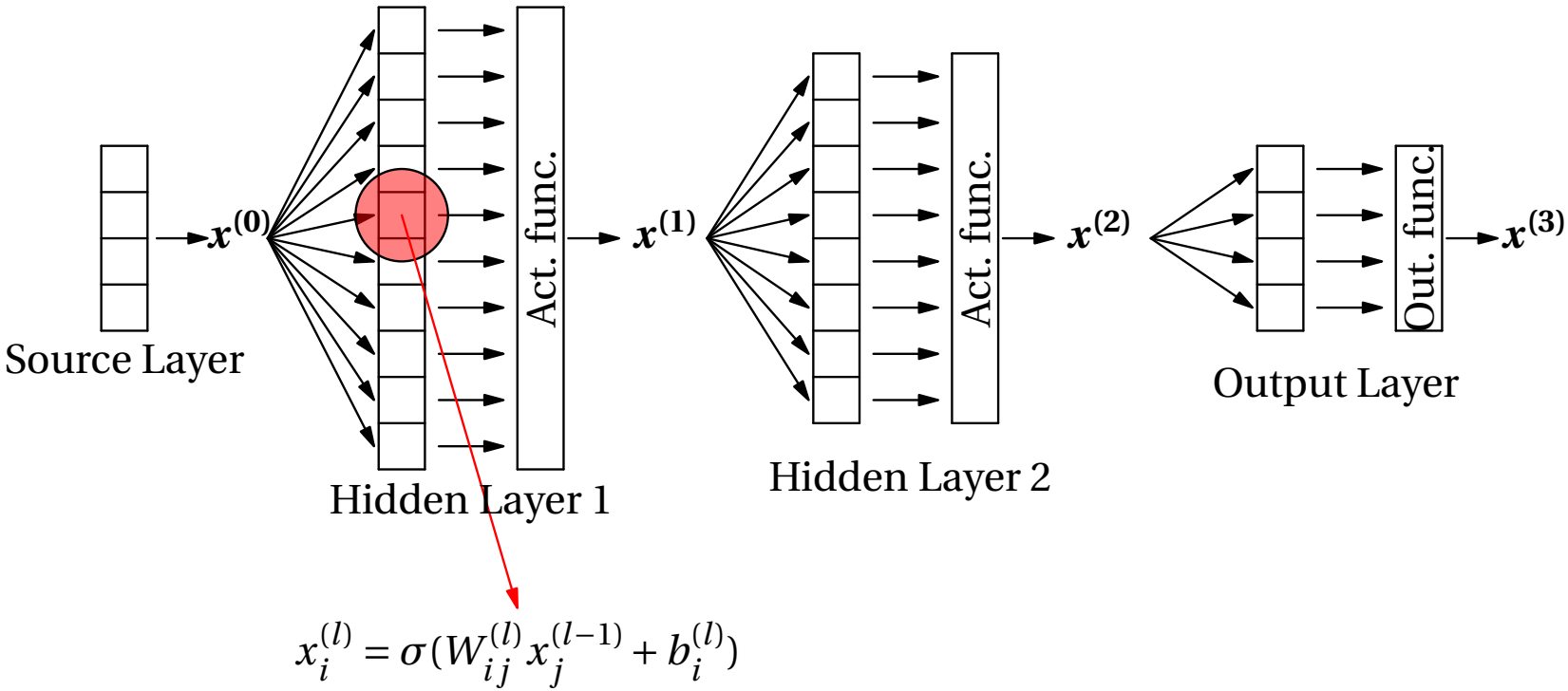


Figure 1: MLP / NN with 2 hidden layers

Operations in MLP / NN

- Consider a MLP / NN of L hidden layers, with the width of layer (l) denoted as H_l, $\forall\, l = 0, 1, ..., L+1$
- For consistency with $\mathbf{f}$ that we are trying to approximate, we must have $H_0 = d$ and $H_{L+1} = D$
- The output vector for l-th layer is denoted by $\mathbf{x}^{(l)} \in \mathbb{R}^{H_l}$, which will be the input to the next layer
- Set $\mathbf{x}^{(0)} = \mathbf{x} \in \mathbb{R}^d$, which will be the input signal provided by the input layer
- In each layer l, $1 \leqslant l \leqslant L+1$, the i-th neuron performs an affine transformation on that layers input $\mathbf{x}^{(l-1)}$ followed by a non-linear transformation

$$x_i^{(l)} = \sigma\Big(\underbrace{W_{ij}^{(l)} x_j^{(l-1)}}_{\text{Einstein sum}} + b_i^{(l)} \Big), \quad 1 \leqslant i \leqslant H_l, \quad 1 \leqslant j \leqslant H_{l-1}$$

- $W_{ij}^{(l)}$, $b_i^{(l)}$: the **weights** and **bias** associated with i-th neuron of layer l

- $\sigma(.)$: the **activation function** which plays a pivotal role in helping the network to represent non-linear complex functions

- Set $\mathbf{W}^{(l)} \in \mathbb{R}^{H_{l-1} \times H_l}$ to be the weight matrix for layer l and $\mathbf{b}^{(l)} \in \mathbb{R}^{H_l}$ to be the bias vector for layer l, one can rewrite the action of the whole layer as

$$\mathbf{x}^{(l)} = \sigma\left(\mathcal{A}^{(l)}(\mathbf{x}^{(l-1)})\right), \quad \mathcal{A}^{(l)}(\mathbf{x}^{(l-1)}) = \mathbf{W}^{(l)}\mathbf{x}^{(l-1)} + \mathbf{b}^{(l)} \quad (1)$$

  where the activation function is applied component-wise

- The action of the network $\mathcal{F} : \mathbb{R}^d \mapsto \mathbb{R}^D$ can be seen as a composition of alternating affine transformations and component-wise activations

$$\mathcal{F}(\mathbf{x}) = \mathcal{A}^{(L+1)} \circ \sigma \circ \mathcal{A}^{(L)} \circ \sigma \circ \mathcal{A}^{(L-1)} \circ \cdots \circ \sigma \circ \mathcal{A}^{(1)}(\mathbf{x}). \quad (2)$$

- The parameters of the network is all the weights and biases, which will be denoted as $\boldsymbol{\theta} = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L+1} \in \mathbb{R}^{N_\theta}$, where $N_\theta = \sum_{l=1}^{L+1}(H_{l-1}+1)H_l$ denotes the total number of parameters of the network

- The network $\mathcal{F}(\mathbf{x};\boldsymbol{\theta})$ represents a family of parameterized functions, where $\boldsymbol{\theta}$ needs to suitably chosen such that the network approximates the target function $f(\mathbf{x})$ at the input $\mathbf{x}$

Universal approximation results

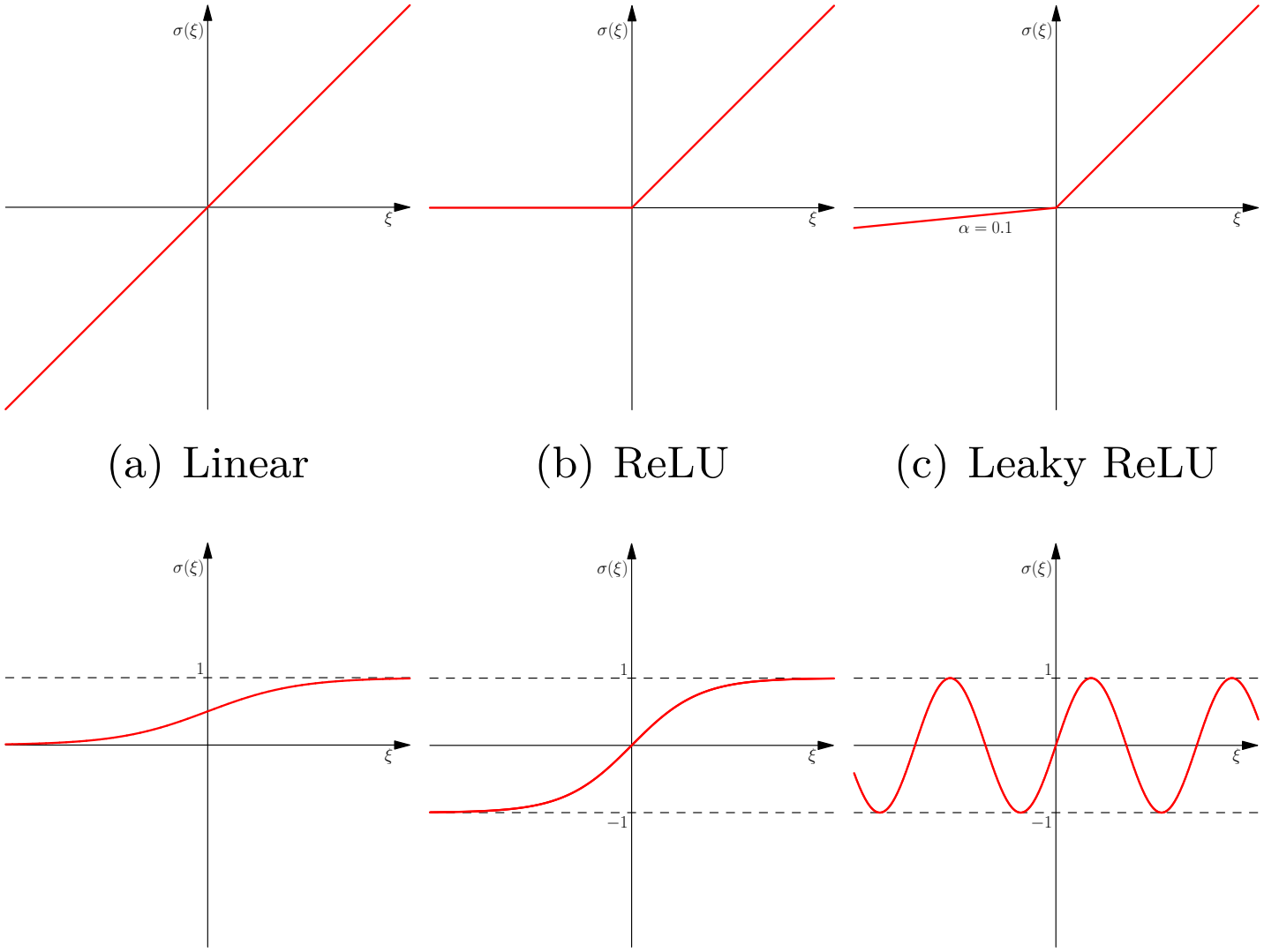
Assume $K \subset \mathbb{R}^d$ is closed and bounded.

Theorem 1 (Pinkus (1999)). Let $f : K \to \mathbb{R}$, i.e., $D = 1$, be a continuous function. Then given an $\epsilon > 0$, there exists an MLP with a single hidden layer ($L = 1$), arbitrary width H and a non-polynomial continuous activation σ such that

$$\max_{\mathbf{x} \in K} |\mathcal{F}(\mathbf{x}; \boldsymbol{\theta}) - f(\mathbf{x})| \leqslant \epsilon.$$

Theorem 2 (Kidger and Lyons (2020)). Let $\mathbf{f} : K \to \mathbb{R}^D$ be a continuous vector-valued function. Then given an $\epsilon > 0$, there exists an MLP with arbitrary number of hidden layers L, each having width $H \geqslant d + D + 2$, a continuous activation σ (with some additional mild conditions), such that

$$\max_{\mathbf{x} \in K} \|\mathcal{F}(\mathbf{x}; \boldsymbol{\theta}) - \mathbf{f}(\mathbf{x})\| \leqslant \epsilon.$$



(a) Linear (b) ReLU (c) Leaky ReLU

(d) Logistic (e) Tanh (f) Sine

Figure 2: Examples of activation functions

Theorem 3 (Yarotsky and Zhevnerchuk (2019)). Let $f : K \to \mathbb{R}$ be a function with two continuous derivates, i.e., $f \in C^2(K)$. Consider an MLP with ReLU activations and $H \geqslant 2d + 10$. Then there exists a network with this configuration such that the error converges as

$$\max_{\mathbf{x} \in K} |\mathcal{F}(\mathbf{x}; \boldsymbol{\theta}) - \mathbf{f}(\mathbf{x})| \leqslant C(N_\theta)^{-4}$$

where C is a constant depending on the number of network parameters.

Training, Validation and Testing of NN

Let us assume that we are given a dataset of pairwise samples $\mathcal{S} = \{(\mathbf{x}_i, \mathbf{y}_i) : 1 \leqslant i \leqslant N\}$ corresponding to a target function $\mathbf{f} : \mathbf{x} \to \mathbf{y}$. We wish to approximate this function using the neural network $\mathcal{F}(\mathbf{x}; \boldsymbol{\theta}, \boldsymbol{\Theta})$ where $\boldsymbol{\theta}$ are the network parameters defined before, while $\boldsymbol{\Theta}$ corresponds to the **hyper-parameters** of the network such as the depth $L+1$, width H, type of activation function σ, etc. The strategy to design a robust network involves three steps:

1. Find the optimal values of $\boldsymbol{\theta}$ (for a fixed $\boldsymbol{\Theta}$) in the **training** phase.

2. Find the optimal values of $\boldsymbol{\Theta}$ in the **validation** phase.

3. Test the performance of the network on unseen data on the **testing** phase.

To accomplish these three tasks, it is first customary to split the dataset $\mathcal{S}$ into three distinct parts: a **training set** with N_{train} samples,

a **validation set** with N_{val} samples and **test set** with N_{test} samples, with $N = N_{\text{train}} + N_{\text{val}} + N_{\text{test}}$. Typically, one uses around 60% of the samples as training samples, 20% as validation samples and the remaining 20% for testing.

Splitting the dataset is necessary as neural networks are heavily over-parameterized functions. The large number of degrees of freedom available to model the data can lead to over-fitting the data. This happens when the error or noise present in the data drives the behavior of the network more than the underlying input-output relation itself. Thus, a part of the data is used to determine $\boldsymbol{\theta}$, and another part to determine the hyper-parameters $\boldsymbol{\Theta}$. The remainder of the data is kept aside for testing the performance of the trained network on unseen data, i.e., the network's ability to **generalize** well.

Now let us discuss how this split is used during the three phases in further details:

Training: Training the network makes use of the training set $\mathcal{S}_{\text{train}}$ to find $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Pi_{\text{train}}(\boldsymbol{\theta})$ where

$$\Pi_{\text{train}}(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{\substack{i=1 \\ (\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{S}_{\text{train}}}}^{N_{\text{train}}} \|\mathbf{y}_i - \mathcal{F}(\mathbf{x}_i; \boldsymbol{\theta}, \boldsymbol{\Theta})\|^2$$

for some fixed $\boldsymbol{\Theta}$. The optimal $\boldsymbol{\theta}^*$ is obtained using a suitable gradient based algorithm (will be discussed later). The function Π_{train} is referred to as the loss function. In the example above we have used the mean-squared loss function. Later we will consider other types of loss functions.

Validation: Validation of the network involves using the validation set $\mathcal{S}_{\text{val}}$ to find $\boldsymbol{\Theta}^* = \arg\min_{\boldsymbol{\Theta}} \Pi_{\text{val}}(\boldsymbol{\Theta})$ where

$$\Pi_{\text{val}}(\boldsymbol{\Theta}) = \frac{1}{N_{\text{val}}} \sum_{\substack{i=1 \\ (\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{S}_{\text{val}}}}^{N_{\text{val}}} \|\mathbf{y}_i - \mathcal{F}(\mathbf{x}_i; \boldsymbol{\theta}^*, \boldsymbol{\Theta})\|^2$$

The optimal $\boldsymbol{\Theta}^*$ is obtained using a techniques such as (random or tensor) grid search.

Testing: Once the "best" network is obtained, characterized by $\boldsymbol{\theta}^*$ and $\boldsymbol{\Theta}^*$, it is evaluated on the test set $\mathcal{S}_{\text{test}}$ to estimate the networks performance on data not used during the first two phases.

$$\Pi_{\text{test}} = \frac{1}{N_{\text{test}}} \sum_{\substack{i=1 \\ (\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{S}_{\text{test}}}}^{N_{\text{test}}} \|\mathbf{y}_i - \mathcal{F}(\mathbf{x}_i; \boldsymbol{\theta}^*, \boldsymbol{\Theta}^*)\|^2.$$

This testing error is also known as the (approximate) **generalizing error** of the network.

Example 1. Let us consider an MLP where all hyper-parameters are fixed except for the following flexible choices: $\sigma \in \{\text{ReLU}, \tanh\}, \quad L \in \{10, 20\}$. We use the following algorithm

1. For each possible σ, L pair:
   (a) Find $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Pi_{\text{train}}(\boldsymbol{\theta})$
   (b) With this $\boldsymbol{\theta}^*$, evaluate $\Pi_{\text{val}}(\boldsymbol{\Theta})$
2. Select $\boldsymbol{\Theta}^*$ to be the one that gave the smallest value of $\Pi_{\text{val}}(\boldsymbol{\Theta})$.
3. Finally, report Π_{test} for this $\boldsymbol{\Theta}^*$ and the corresponding $\boldsymbol{\theta}^*$.

Generalizability

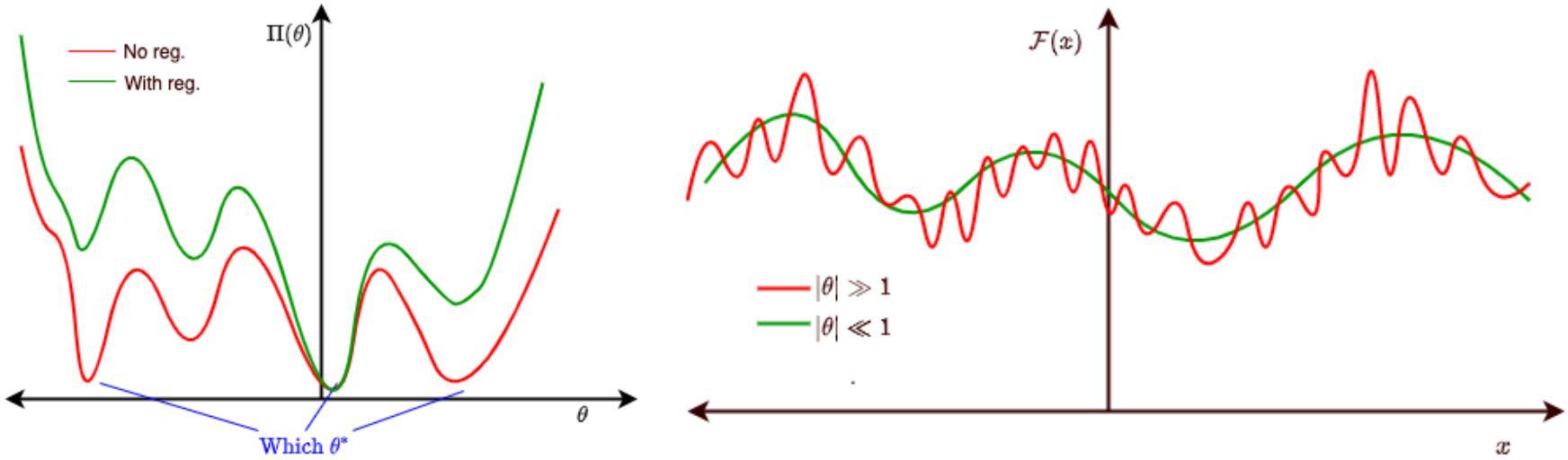
If we train a network that has a small value of Π_{train} and Π_{val}, does it ensure that Π_{test} will be small? This question is addressed by study-

ing the **generalizability** of the trained network, i.e., it capability to perform well on data not seen while training/validating the network. If the network is trained to overfit the training data, the network will typically lead to poor predictions on test data. Typically, if $\mathcal{S}_{\text{train}}$, $\mathcal{S}_{\text{val}}$ and $\mathcal{S}_{\text{test}}$ are chosen from the same distribution of data, then a small value of $\Pi_{\text{train}}, \Pi_{\text{val}}$ can lead to small values of Π_{test}. Let us look at the commonly used technique to avoid data overfitting, called **regularization**.

Regularization

Neural networks are almost always **over-parametrized**, i.e., $N_\theta \gg N$ where N is the number of training samples. This would lead to a highly non-linear network model, for which the loss function $\Pi(\boldsymbol{\theta})$ (where we omit the subscript "train" for brevity) can have a landscape with many local minimas (see Figure 3(a)). Then how do we determine which minima leads to a better generalization? To nudge the choice of $\boldsymbol{\theta}^*$ in a more favorable direction, a regularization technique

can be employed.



(a) Loss function landscape (b) Network sensitivity

Figure 3: The effect of regularization on the loss function. We have assumed a scalar θ for easier illustration.

The simplest method of regularization involves augmenting a penalty

term to the loss function:

$$\Pi(\boldsymbol{\theta}) \longrightarrow \Pi(\boldsymbol{\theta}) + \alpha \, \|\boldsymbol{\theta}\|, \quad \alpha \geqslant 0$$

where α is a regularization hyper-parameter, and $\|\boldsymbol{\theta}\|$ is a suitable norm of the network parameters $\boldsymbol{\theta}$. This augmentation can change the landscape of $\Pi(\boldsymbol{\theta})$ as illustrated in Figure 3(a). In other words, such a regularization encourages the selection of a minima corresponding to smaller values of the parameters $\boldsymbol{\theta}$.

It is not obvious why a smaller value of $\boldsymbol{\theta}$ would be a better choice. To see why this is better, consider the intermediate network output

$$x_1^{(1)} = \sigma(W_{1j}^{(1)} x_j^{(0)} + b_1^{(1)}),$$

which gives

$$\frac{\partial x_1^{(1)}}{\partial x_1^{(0)}} = \sigma'(W_{1j}^{(1)} x_j^{(0)} + b_1^{(1)}) W_{11}^{(1)} \propto W_{11}^{(1)}.$$

Since this derivate scales with $W_{11}^{(1)}$, this implies that $\left|\frac{\partial \mathcal{F}(\mathbf{x})}{\partial x_1^{(0)}}\right|$ scales with $W_{11}^{(1)}$ as well. If $|W_{11}^{(1)}| \gg 1$, then network would be very sensitive to even small changes in the input $x_1^{(0)}$, i.e., the network would be ill-posed. As illustrated in Figure 3(b), using a proper regularization would help avoid over fitting.

Let us consider some common types of regularization:

- l_2 **regularization**: Here we use the l_2 norm in the regularization term

$$\|\boldsymbol{\theta}\| = \|\boldsymbol{\theta}\|_2 = \left(\sum_{i=1}^{N_\theta} \theta_i^2\right)^{1/2}.$$

- l_1 **regularization**: Here we use the l_1 norm in the regularization term

$$\|\boldsymbol{\theta}\| = \|\boldsymbol{\theta}\|_1 = \sum_{i=1}^{N_\theta} |\theta_i|,$$

which promotes the sparsity of $\boldsymbol{\theta}$.

Gradient descent

Recall that we wish to solve the minimization problem $\boldsymbol{\theta}^* = \arg\min \Pi(\boldsymbol{\theta})$ in the training phase. This minimization problem can be solved using gradient descent (GD), also known as steepest descent. Consider the Taylor expansion about $\boldsymbol{\theta}_0$

$$\Pi(\boldsymbol{\theta}_0 + \Delta\boldsymbol{\theta}) = \Pi(\boldsymbol{\theta}_0) + \frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_0) \cdot \Delta\boldsymbol{\theta} + \frac{\partial^2 \Pi}{\partial \theta_i \partial \theta_j}(\hat{\boldsymbol{\theta}}) \Delta\theta_i \Delta\theta_j$$

for some $\hat{\boldsymbol{\theta}}$ in a small neighbourhood of $\boldsymbol{\theta}_0$. When $|\Delta\boldsymbol{\theta}|$ is small and assuming $\frac{\partial^2 \Pi}{\partial \theta_i \partial \theta_j}$ is bounded, we can neglect the second order term and just consider the approximation

$$\Pi(\boldsymbol{\theta}_0 + \Delta\boldsymbol{\theta}) \approx \Pi(\boldsymbol{\theta}_0) + \frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_0) \cdot \Delta\boldsymbol{\theta}.$$

In order to lower the value of the loss function as much as possible compared to its evaluation at $\boldsymbol{\theta}_0$, i.e. minimize $\Delta\Pi = \Pi(\boldsymbol{\theta}_0 + \Delta\boldsymbol{\theta}) - \Pi(\boldsymbol{\theta}_0)$, we need to choose the step $\Delta\boldsymbol{\theta}$ in the opposite direction of the gradient, i.e.:

$$\Delta\boldsymbol{\theta} = -\eta \frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_0)$$

with the step-size $\eta \geqslant 0$, also known as the **learning-rate**. This is yet another hyper-parameter that we need to tune during the validation phase. This is the crux of the GD algorithm, and can be summarized as follows:

1. Initialize $k = 0$ and $\boldsymbol{\theta}_0$
2. While $|\Pi(\boldsymbol{\theta}_k)| > \epsilon_1$, do
   (a) Evaluate $\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k)$
   (b) Update $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k)$
   (c) Increment $k = k + 1$

Convergence: Assume that $\Pi(\boldsymbol{\theta})$ is convex and differentiable, and its gradient is Lipschitz continuous with Lipschitz constant $\mathcal{K}$. Then for a $\eta \leqslant 1/\mathcal{K}$, the GD updates converges as

$$\|\boldsymbol{\theta}^* - \boldsymbol{\theta}_k\|_2 \leqslant \frac{C}{k}.$$

However, in most scenarios $\Pi(\boldsymbol{\theta})$ is not convex. If there is more than one minima, then what kind of minima does GD like to pick? To answer this, consider the loss function for a scalar θ as shown in Figure 4, which has two valleys. Let's assume that the profile of $\Pi(\theta)$ in the each valley can be approximated by a (centered) parabola

$$\Pi(\theta) \approx \frac{1}{2} a\theta^2$$

where $a > 0$ is the curvature of each valley. Note that the curvature of the left valley is much smaller than the curvature of the right valley. Let's pick a constant learning rate η and a starting value θ_0 in either

of the valleys. Then,

$$\frac{\partial \Pi}{\partial \theta}(\theta_0) = a\theta_0$$

and the new point after a GD update will be $\theta_1 = \theta_0(1-a\eta)$. Similarly, it is easy to see that all subsequent iterates write $\theta_{k+1} = \theta_k(1-a\eta)$. For convergence, we need

$$\left|\frac{\theta_{k+1}}{\theta_k}\right| < 1 \quad \implies |1-a\eta| < 1.$$

Since $a > 0$ in the valleys, we will need the following condition on the learning rate

$$-1 < 1 - a\eta \implies a\eta < 2.$$

If we fix η, then for convergence we need the local curvature to satisfy $a < 2/\eta$. In other words, GD will prefer to converge to a minima with a flat/small curvature, i.e., it will prefer the minima in the left valley.

If the starting point is in the right valley, there is a chance that we will keep overshooting the right minima and bounce off the opposite wall till the GD algorithm slingshots θ_k outside the valley. After this it will enter the left valley with a smaller curvature and gradually move towards its minima.

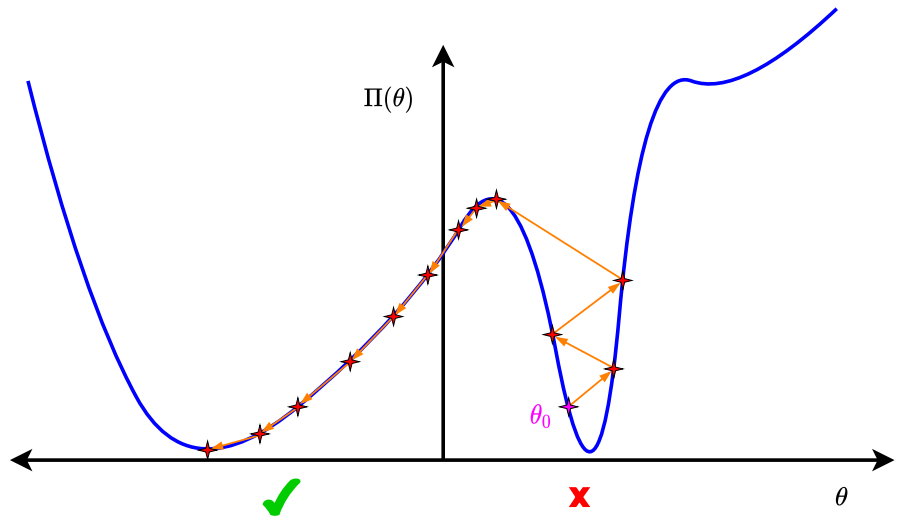


Figure 4: GD prefers flatter minimas.

While it is clear that GD prefers flat minima, what is not clear is why are flat minima better. There is empirical evidence that the

parameter values obtained at flat minima tend to generalize better, and therefore are to be preferred.

Some advanced optimization algorithms

We discussed how GD can be used to solve the optimization problem involved in training neural networks. Let us look at a few advanced and popular optimization techniques motivated by GD.

In general, the update formula for most optimization algorithms make use of the following formula

$$[\boldsymbol{\theta}_{k+1}]_i = [\boldsymbol{\theta}_k]_i - [\eta_k]_i [\mathbf{g}_k]_i, \quad 1 \leqslant i \leqslant N_\theta, \tag{3}$$

where $[\eta_k]_i$ is the component-wise learning rate and the vector-valued function $\mathbf{g}$ depends/approximates the gradient. Note that the notation $[.]_i$ is used to denote the i-th component of the vector. Also note that the learning rate is allowed to depend on the iteration number

k. The GD method makes use of

$$[\eta_k]_i = \eta, \quad \mathbf{g}_k = \frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k).$$

An issue with the GD method is that the convergence to the minima can be quite slow if η is not suitably chosen. For instance, consider the objective function landscape shown in Figure 5, which has sharper gradients along the $[\theta]_2$ direction compared to the $[\theta]_1$ direction. If we start from a point, such as the one shown in the figure, then if η is too large (but still within the stable bounds) the updates will keep zig-zagging its way towards the minima. Ideally, for the particular situation shown in Figure 5, we would like the steps to take longer strides along the $[\theta]_1$ compared to the $[\theta]_2$ direction, thus reaching the minima faster.

Let us look at two popular methods that are able to overcome some of the issues faced by GD.

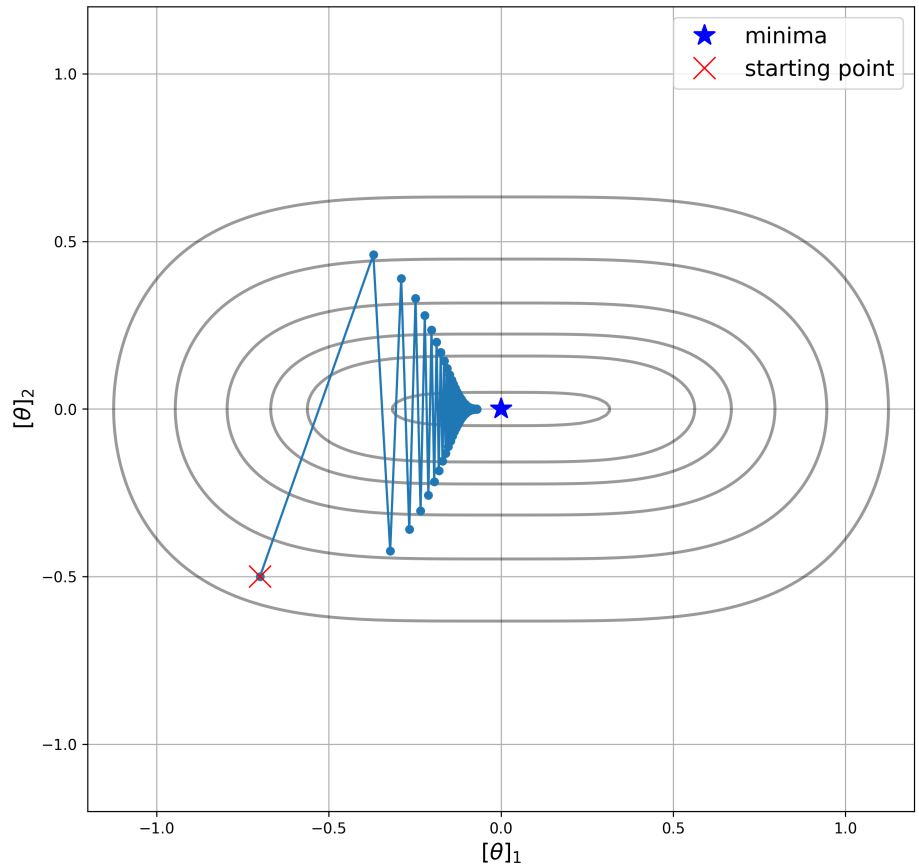


Figure 5: Zig-zagging updates with GD.

Momentum methods

Momentum methods make use of the history of the gradient, instead of just the gradient at the previous step. The formula for the update is given by

$$[\eta_k]_i = \eta, \quad \mathbf{g}_k = \beta_1 \mathbf{g}_{k-1} + (1-\beta_1)\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k), \quad \mathbf{g}_{-1} = 0$$

where $\mathbf{g}_k$ is a weighted moving average of the gradient. This weighting is expected to smoothen out the zig-zagging seen in Figure 5 by cancelling out the components of gradient along the $[\theta]_2$ direction and move more smoothly towards the minima. A commonly used value for β_1 is 0.9.

Adam

The Adam optimization was introduced by Kingma and Ba Kingma and Ba (2017), which makes use of the history of the gradient as

well the second moment (which is a measure of the magnitude) of the gradient. For an initial learning rate η, the updates are given by

$$\begin{aligned}
\mathbf{g}_k &= \beta_1 \mathbf{g}_{k-1} + (1-\beta_1)\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k) \\
[\mathbf{G}_k]_i &= \beta_2 [\mathbf{G}_{k-1}]_i + (1-\beta_2)\left(\frac{\partial \Pi}{\partial \boldsymbol{\theta}_i}(\boldsymbol{\theta}_k)\right)^2 \\
[\eta_k]_i &= \frac{\eta}{\sqrt{[\mathbf{G}_k]_i} + \epsilon}
\end{aligned} \tag{4}$$

where $\mathbf{g}_k$ and $\mathbf{G}_k$ are the weighted running averages of the gradients and the square of the gradients, respectively. The recommended values for the hyper-parameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. Note that the learning rate for each component is different. In particular, the larger the magnitude of the gradient for a component the smaller is its learning rate. Referring back to the example in Figure 5, this would mean a smaller learning rate for θ_2 in comparison to θ_1, and therefore will help alleviate the zig-zag path of the optimization algorithm.

Remark 1. The Adam algorithm also has additional correction steps for $\mathbf{g}_k$ and $\mathbf{G}_k$ to improve the efficiency of the algorithm. See Kingma and Ba (2017) for details.

Stochastic optimization

We note that the training loss can be rewritten as

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \Pi_i(\boldsymbol{\theta}), \quad \Pi_i(\boldsymbol{\theta}) = \|\mathbf{y}_i - \mathcal{F}(\mathbf{x}_i; \boldsymbol{\theta}, \boldsymbol{\Theta})\|^2$$

Thus, the gradient of the loss function is

$$\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \frac{\partial \Pi_i}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta})$$

However, taking the summation of gradients can be very expensive since N_{train} is typically very large, $N_{\text{train}} \sim 10^6$. One easy way to

circumvent this problem is to use the following update formula (shown here for the GD method)

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \frac{\partial \Pi_i}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k), \tag{5}$$

where i is randomly chosen for each update step k. This is known as **stochastic gradient descent**. Remarkably, this modified algorithm does converge assuming that $\Pi_i(\boldsymbol{\theta})$ is convex and differentiable, and $\eta_k \sim 1/\sqrt{k}$ (Nemirovski et al. (2009)). To illustrate why η_k needs to decay, consider the toy function(s) for $\boldsymbol{\theta} \in \mathbb{R}^2$

$$\begin{aligned}
\Pi_1(\boldsymbol{\theta}) &= ([\theta]_1 - 1)^2 + ([\theta]_2 - 1)^2 \\
\Pi_2(\boldsymbol{\theta}) &= ([\theta]_1 + 1)^2 + 0.5\,([\theta]_2 - 1)^2 \\
\Pi_3(\boldsymbol{\theta}) &= 0.7\,([\theta]_1 + 1)^2 + 0.5\,([\theta]_2 + 1)^2 \\
\Pi_4(\boldsymbol{\theta}) &= 0.7\,([\theta]_1 - 1)^2 + \frac{1}{2}\,([\theta]_2 + 1)^2 \\
\Pi(\boldsymbol{\theta}) &= \frac{1}{4}(\Pi_1(\boldsymbol{\theta}) + \Pi_2(\boldsymbol{\theta}) + \Pi_3(\boldsymbol{\theta}) + \Pi_4(\boldsymbol{\theta})).
\end{aligned} \tag{6}$$

The contour plots of these functions in shown in Figure 6(a), where the black contour plots corresponds to $\Pi(\boldsymbol{\theta})$. Note that the $\boldsymbol{\theta}^* = (0, 0)$ is the unique minima for $\Pi(\boldsymbol{\theta})$. We consider solving with the SGD algorithm with a constant learning rate $\eta_k = 0.4$ and a decaying learning rate $\eta_k = 0.4/\sqrt{k}$. Starting with $\boldsymbol{\theta}_0 = (-1.0, 2.0)$ and randomly selecting $i \in 1, 2, 3, 4$ for each step k, we run the algorithm for 10,000 iterations. The first 10 steps with each learning rate is plotted in Figure 6(a). We can clearly see that without any decay in the learning rate, the SGD algorithm keeps overshooting the minima. In fact, this behaviour continues for all future iterations as can be seen in Figure 6(b) where the norm of the updates does not decay (we expect it to decay to $|\boldsymbol{\theta}^*| = 0$). On the other hand, we quickly move closer to $\boldsymbol{\theta}^*$ if the learning rate decays as $1/\sqrt{k}$.

The reason for reducing the step size as we approach closer to the minima is that far away from the minima for Π the gradient vector for Π and all the individual Π_i's align quite well. However, as we approach closer to the minima for Π this is not the case and therefore one is required to take smaller steps so as not be thrown off to a region

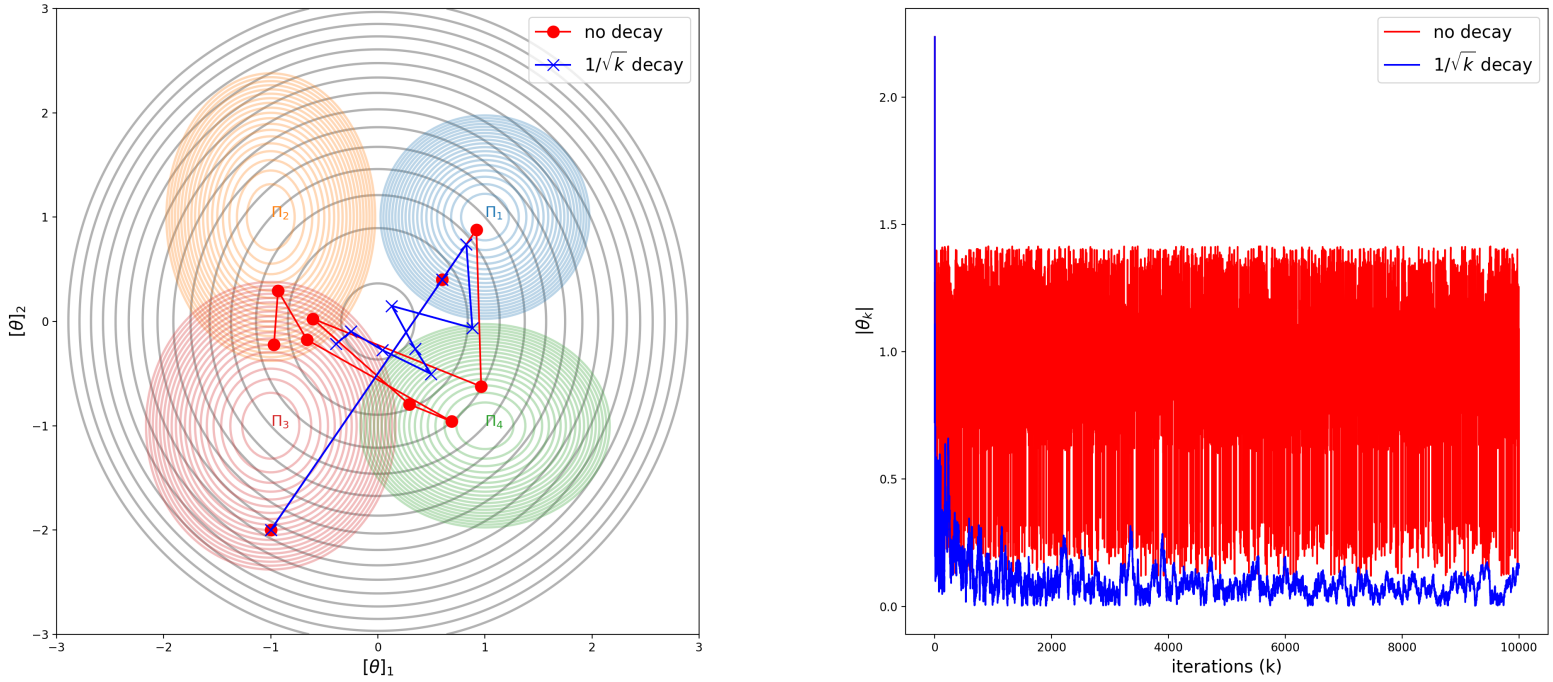
far away from the minima.

In practice, stochastic optimization algorithms are not used for the following reasons:

1. Although the loss function decays with the number of iterations, it fluctuates in a chaotic manner close the the minima and never manages to reach the minima.

2. While handling all samples at once can be computationally expensive, handling a single sample at a time severly under-utilizes the computational and memory resources.

However, a compromise can be made by using **mini-batch optimization**. In this strategy, the dataset of N_{train} samples is split into N_{batch} disjoint subsets known as mini-batches. Each mini-batch contains $\overline{N}_{\text{train}} = N_{\text{train}}/N_{\text{batch}}$ samples, which also refered to as the batch-size. Thus, the gradient of the loss function can be approxi-



(a) Function contours and paths

(b) Norm of updates

Figure 6: SGD algorithm with and without a decay in the learning rate.

mated by

$$\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \frac{\partial \Pi_i}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}) \approx \frac{1}{\overline{N}_{\text{train}}} \sum_{i \in \text{batch}(j)} \frac{\partial \Pi_i}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}). \tag{7}$$

Note that taking $N_{\text{batch}} = 1$ leads to the original optimization algorithms, while take $N_{\text{batch}} = N_{\text{train}}$ gives the stochastic gradient descent algorithm. One typically chooses a batch-size to maximize the amount of data that can be loaded into the RAM at one time. We define an **epoch** as one full pass through all samples (or mini-batches) of the full training set. The following describes the mini-batch stochastic optimizati algorithm:

1. For epoch = 1, ..., J
   (a) Randomly shuffle the full training set
   (b) Create N_{batch} mini-batches
   (c) For $i = 1, \cdots, N_{\text{batch}}$

i. Evaluate the batch gradient using (7).

ii. Update $\boldsymbol{\theta}$ using this gradient and your favorite optimization algorithm (gradient descent, momentum, or Adam).

Remark 2. There is an interesting study Wu et al. (2018) that suggests that stochastic gradient descent might actually help in selecting minima that generalize better. In that study the authors prove that SGD prefers minima whose curvature is more homogeneous. That is, the distribution of the curvature of each of the components of the loss function is sharp and centered about a small value. This is contrast to minima where the overall curvature might be small; however the distribution of the curvature of each component of loss function is more spread out. Then they go on to show (empirically) that the more homogeneous minima tend to generalize better than their heterogeneous counterparts.

Calculating gradients using back-propagation

The final piece of the training algorithm that we need to understand is how the gradients are actually evaluated while training the network. Recall the output $\mathbf{x}^{(l+1)}$ of layer $l+1$ is given by

$$\text{Affine transform:} \quad \xi_i^{(l+1)} = W_{ij}^{(l+1)} x_j^{(l)} + b_i^{(l+1)}, \quad 1 \leqslant i \leqslant H_{l+1} \tag{8}$$

$$\text{Non-linear transform:} \quad x_i^{(l+1)} = \sigma\left(\xi_i^{(l+1)}\right), \quad 1 \leqslant i \leqslant H_{l+1}. \tag{9}$$

Given a training sample $(\mathbf{x}, \mathbf{y})$, set $\mathbf{x}^{(0)} = \mathbf{x}$. The value of the loss/objective function (for this particular sample) can be evaluated using the forward pass:

1. For $l = 1, ..., L+1$
   (a) Evaluate $\xi^{(l)}$ using (8).
   (b) Evaluate $\mathbf{x}^{(l)}$ using (9).

2. Evaluate the loss function for the given sample

$$\Pi(\boldsymbol{\theta}) = \|\mathbf{y} - \mathcal{F}(\mathbf{x}; \boldsymbol{\theta}, \boldsymbol{\Theta})\|^2.$$

This operation can be written succinctly in the form of a computational graph as shown in Figure 7. In this figure, the lower portion of the graph represents the evaluation of the loss function Π.

We would of course need to repeat this step for all samples in the training set (or a mini-batch for stochastic optimization). For simplicity, we restrict the discussion to the evaluation of the loss and its gradient for a single sample.

In order to update the network parameters, we need $\frac{\partial \Pi}{\partial \boldsymbol{\theta}}$, or more precisely $\frac{\partial \Pi}{\partial \mathbf{W}^{(l)}}$, $\frac{\partial \Pi}{\partial \mathbf{b}^{(l)}}$ for $1 \leqslant l \leqslant L+1$. We will derive expressions for these derivatives by first deriving expressions for $\frac{\partial \Pi}{\partial \xi^{(l)}}$ and $\frac{\partial \Pi}{\partial \mathbf{x}^{(l)}}$.

From the computational graph it is easy to see how each hidden variable in the network is transformed to the next. Recognizing this,

and applying the chain rule repeatedly yields

$$\frac{\partial \Pi}{\partial \xi^{(l)}} = \frac{\partial \Pi}{\partial \mathbf{x}^{(L+1)}} \cdot \frac{\partial \mathbf{x}^{(L+1)}}{\partial \xi^{(L+1)}} \cdot \frac{\partial \xi^{(L+1)}}{\partial \mathbf{x}^{(L)}} \cdots \frac{\partial \mathbf{x}^{(l+1)}}{\partial \xi^{(l+1)}} \cdot \frac{\partial \xi^{(l+1)}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial \mathbf{x}^{(l)}}{\partial \xi^{(l)}}. \quad (10)$$

In order to evaluate this expression we need to evaluate the following terms:

$$\frac{\partial \Pi}{\partial \mathbf{x}^{(L+1)}} = -2(\mathbf{y} - \mathbf{x}^{(L+1)})^T \quad (11)$$

$$\frac{\partial \xi^{(l+1)}}{\partial \mathbf{x}^{(l)}} = \mathbf{W}^{(l+1)} \quad (12)$$

$$\frac{\partial \mathbf{x}^{(l)}}{\partial \xi^{(l)}} = \mathbf{S}^{(l)} \equiv \mathrm{diag}[\sigma'(\xi_1^{(l)}), \cdots, \sigma'(\xi_{H_l}^{(l)})], \quad (13)$$

where the last two relations hold for any network layer l, H_l is the width of that particular layer, and σ' denotes the derivative of the activation with respect to its argument. Using these relations in (10),

we arrive at,

$$\frac{\partial \Pi}{\partial \xi^{(l)}} = \frac{\partial \Pi}{\partial \mathbf{x}^{(L+1)}} \cdot \mathbf{S}^{(L+1)} \cdot \mathbf{W}^{(L+1)} \ldots \mathbf{S}^{(l+1)} \cdot \mathbf{W}^{(l+1)} \cdot \mathbf{S}^{(l)}. \tag{14}$$

Taking the transpose, and recognizing that $\Sigma^{(l)}$ is diagonal and therefore symmetric, we finally arrive at

$$\frac{\partial \Pi}{\partial \xi^{(l)}} = \mathbf{S}^{(l)} \mathbf{W}^{(l+1)T} \mathbf{S}^{(l+1)} \ldots \mathbf{W}^{(L+1)T} \mathbf{S}^{(L+1)} [-2(\mathbf{y} - \mathbf{x}^{(L+1)})]. \tag{15}$$

This evaluation can also be represented as a computational graph. In fact, as shown in Figure 7, it can be appended to the original graph, where this part of the computation appear in the upper row of the graph. Note that we are now traversing in the backward direction. Hence the name back propagation.

The final step is to evaluate an explicit expression for $\frac{\partial \Pi}{\partial \mathbf{W}^{(l)}}$. This can be done by recognizing,

$$\frac{\partial \Pi}{\partial \mathbf{W}^{(l)}} = \frac{\partial \Pi}{\partial \xi^{(l)}} \cdot \frac{\partial \xi^{(l)}}{\partial \mathbf{W}^{(l)}} = \frac{\partial \Pi}{\partial \xi^{(l)}} \otimes \mathbf{x}^{(l-1)}, \tag{16}$$

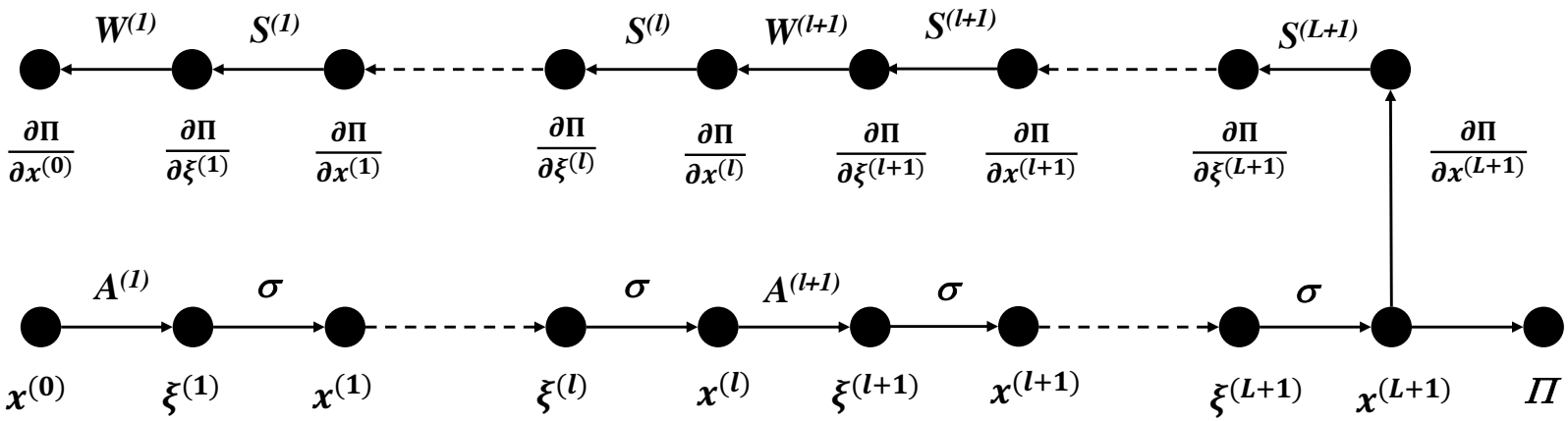


Figure 7: Computational graph for computing the loss function and its derivatives with respect to hidden/latent vectors.

where $[\mathbf{x} \otimes \mathbf{y}]_{ij} = x_i y_j$ is the outer product. Thus, in order to evaluate $\frac{\partial \Pi}{\partial \mathbf{W}^{(l)}}$ we need $\mathbf{x}^{(l-1)}$ which is evaluted during the forward phase and $\frac{\partial \Pi}{\partial \xi^{(l)}}$ which is evaluated during back propagation.

Question 1. Can you derive a similar set of expressions and the corresponding algorithm to evaluate $\frac{\partial \Pi}{\partial \mathbf{b}^{(l)}}$?

Question 2. Can you derive an explicit expression for $\frac{\partial \mathbf{x}^{(L+1)}}{\partial \mathbf{x}^{(0)}}$. That is the an expression for the derivative of the output of the network with respect to its input? This is a very useful quantity that finds use in algorithms like physics informed neural networks and Wasserstein generative adversarial networks.

Regression versus classification

Till now, given the labelled dataset $\mathcal{S} = \{(\mathbf{x}_i, \mathbf{y}_i) : 1 \leqslant i \leqslant N\}$, we have considered two types of losses

- The mean square error (MSE)

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \|\mathbf{y}_i - \mathcal{F}(\mathbf{x}_i; \boldsymbol{\theta}, \boldsymbol{\Theta})\|^2$$

- The mean absolute error (MAE)

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \|\mathbf{y}_i - \mathcal{F}(\mathbf{x}_i; \boldsymbol{\theta}, \boldsymbol{\Theta})\|$$

Neural networks with the above losses can be used to solve various regression problems where the underlying function is highly nonlinear and the inputs/outputs are multi-dimensional.

Example 2. Given the house / apartment features such as the zip code, the number of bedrooms / bathrooms, carpet area, age of construction, etc, predict the outcomes such as the market selling price, or the number of days on the market.

Now let us consider some examples of classification problems, where the output of the network typically lies in a discrete finite set.

Example 3. Given the symptoms and blood markers of patients with COVID-19, predict whether they will need to be admitted to ICU. So the input and output for this problem would be

$$\mathbf{x} = [\text{pulse rate, temperature, } \text{SPO}_2\text{, procalcitonin, ...}]$$
$$\mathbf{y} = [p_1, p_2]$$

where p_1 is the probability of being admitted to the ICU, while p_2 is the probability of not being admitted. Note that $0 \leqslant p_1, p_2 \leq 1$ and $p_1 + p_2 = 1$.

Example 4. Given a set of images of animals, predict whether the animal is a dog, cat or bird. In this case, the input and output should be

$$\mathbf{x} = \text{the image}$$
$$\mathbf{y} = [p_1, p_2, p_2]$$

where p_1, p_2, p_3 is the probability of being a dog, cat or bird, respectively.

Since the output for the classification problem corresponds to probabilities, we need to make a few changes to the network

1. Make use of an output function at the end of the output layer that suitably transforms the output vector into the desired form, i.e, a vector of probabilities. This is typically done using the **softmax function**

$$x_i^{(L+1)} = \frac{\exp\left(\xi_i^{(L+1)}\right)}{\sum_{j=1}^{C} \exp\left(\xi_j^{(L+1)}\right)}$$

where C is the number of classes (and also the output dimension). Verify that with this transformation, the components of the $\mathbf{x}^{(L+1)}$ form a convex combination, i.e., $x_i^{(L+1)} \in [0, 1]$ and $\sum_{i=1}^{C} x_i^{(L+1)} = 1$.

2. The output labels for the various samples need to be one-hot encoded. In other words, for the sample $(\mathbf{x}, \mathbf{y})$, the output label $\mathbf{y}$ should have dimension $D = C$, and whose component is 1 only for the component signifying the class $\mathbf{x}$ belongs to, otherwise 0. For instance, in Example 4

$$\mathbf{y} = \begin{cases} [1,\, 0,\, 0]^\top & \text{if } \mathbf{x} \text{ is a dog,} \\ [0,\, 1,\, 0]^\top & \text{if } \mathbf{x} \text{ is a cat,} \\ [0,\, 0,\, 1]^\top & \text{if } \mathbf{x} \text{ is a pig.} \end{cases}$$

3. Although the MSE or MSA can still be used as the loss function, it is preferable to use the cross-entropy loss function

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \sum_{c=1}^{C} -y_{ci} \log(\mathcal{F}_c(\mathbf{x}_i; \boldsymbol{\theta})), \tag{17}$$

where y_{ci} is the c-th component of the true label for the i-th sample. The loss function in (17) treats y_c and $\mathcal{F}_c$ as probability distributions and measures the discrepancy between the

two. It can be shown to be related to the Kullback-Liebler divergence between the two distributions. Compared to MSE, this loss function severely penalizes strongly confident incorrect predictions. This is demonstrated in Example 5

Example 5. Let us consider a binary classification problem, i.e., $C = 2$. For a given $\mathbf{x}$, let $\mathbf{y} = [0, 1]$ and let the prediction be $\mathcal{F} = [p, 1-p]$. Clearly, a small value of p is preferred. Therefore any reasonable cost function should penalize large values of p. Now let us evaluate the error using various loss functions

- MSE Loss $= (0-p)^2 + (1-1+p)^2 = 2p^2$.
- Cross-entropy Loss $= -(0\log(p) + 1\log(1-p) = -\log(1-p)$.

Note that both losses penalize large values of p. Also when $p = 0$, both losses are zero. However, as $p \to 1$ (which would lead the wrong prediction), the MSE loss $\to 2$, while the cross-entropy loss $\to \infty$. That is, it strongly penalizes incorrect confident predictions.

References

Kidger, P., Lyons, T., 2020. Universal Approximation with Deep Narrow Networks, in: Abernethy, J., Agarwal, S. (Eds.), Proceedings of Thirty Third Conference on Learning Theory, PMLR. pp. 2306–2327. URL: https://proceedings.mlr.press/v125/kidger20a.html.

Kingma, D.P., Ba, J., 2017. Adam: A method for stochastic optimization. https://arxiv.org/abs/1412.6980v9. doi:10.48550/ARXIV.1412.6980.

Nemirovski, A., Juditsky, A., Lan, G., Shapiro, A., 2009. Robust stochastic approximation approach to stochastic programming. SIAM Journal on Optimization 19, 1574–1609. URL: https://doi.org/10.1137/070704277, doi:10.1137/070704277, arXiv:https://doi.org/10.1137/070704277.

Pinkus, A., 1999. Approximation theory of the mlp model in

neural networks. Acta Numerica 8, 143–195. doi:10.1017/S0962492900002919.

Wu, L., Ma, C., E, W., 2018. How sgd selects the global minima in over-parameterized learning: A dynamical stability perspective, in: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (Eds.), Advances in Neural Information Processing Systems, Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper/2018/file/6651526b6fb8f29a00507de6a49ce30f-Paper.pdf.

Yarotsky, D., Zhevnerchuk, A., 2019. The phase diagram of approximation rates for deep neural networks. https://arxiv.org/abs/1906.09477. doi:10.48550/ARXIV.1906.09477.