3010

ARTIFICIAL INTELLIGENCE

Lecture 8 Neural Network Learning

Masashi Shimbo

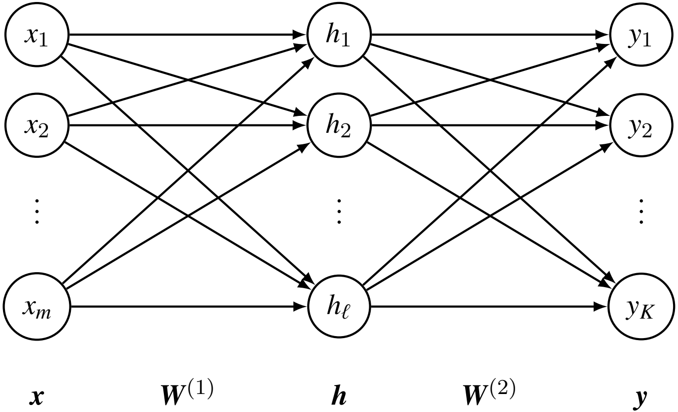
2019-06-03

Based on the lecture slides by Hiroshi Noji

Outline

- **Review: Feed-forward neural networks**
- Learning as optimization
- Popular loss functions
- Stochastic gradient descent (SGD)
- Back propagation

(Two-layer) feed-forward neural networks (NNs)



$$\begin{aligned}\boldsymbol{h} &= g(\boldsymbol{W}^{(1)}\boldsymbol{x})\\ \boldsymbol{y} &= \boldsymbol{W}^{(2)}\boldsymbol{h}\end{aligned}$$

or

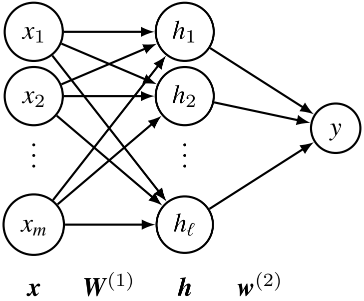
$$\begin{aligned}\boldsymbol{y} &= \overbrace{\boldsymbol{W}^{(2)}g(\boldsymbol{W}^{(1)}}^{NN}\boldsymbol{x})\\ &= NN(\boldsymbol{x})\end{aligned}$$

Let θ denote the set of parameters:

$$\theta = \{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}\}$$

Note on the output layer

- For K-class classification ($K \geq 3$), each y_i is the score indicating how likely $\boldsymbol{x}$ is in class i
  - We use z to denote the label (output)
  - The predicted label is then: $z = \text{argmax}_i \, y_i$
- For binary classification ($K = 2$), we only need one node y.



Output is determined by

$$z = \begin{cases} +1 & \text{if } y \geq 0 \\ -1 & \text{if } y < 0 \end{cases}$$

- y (or y_i) is **score** ($\in \mathbb{R}$), not **label** ($\in \{1, \ldots, K\}$)
- Training data: $\{(\boldsymbol{x}_1, z_1), (\boldsymbol{x}_2, z_2), \ldots, (\boldsymbol{x}_N, z_N)\}$

Outline

- Review: Feed-forward neural networks
- **Learning as optimization**
- Popular loss functions
- Stochastic gradient descent (SGD)
- Back propagation

Learning NN = numerical optimization problem

- $\theta =$ parameters in NN (that we want to "optimize")

  e.g., for two-layer feed-forward NNs, $\theta = \{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}\}$

- Define a **local loss function** $\ell(\boldsymbol{x}, z, \theta)$

  ➥ small $\ell(\boldsymbol{x}, z, \theta)$ indicates parameter θ works well on example $(\boldsymbol{x}, z)$

- Given training data D, the **total loss** (or simply **loss function**) L is defined as

$$L(D, \theta) = \sum_{(\boldsymbol{x},z)\in D} \ell(\boldsymbol{x}, z, \theta)$$

➥ Objective = find "optimal" θ that **minimizes** $L(D, \theta)$

Learning NN = numerical optimization problem

- $\theta =$ parameters in NN (that we want to "optimize")

  e.g., for two-layer feed-forward NNs, $\theta = \{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}\}$

- Define a **local loss function** $\ell(\boldsymbol{x}, z, \theta)$

  ➥ small $\ell(\boldsymbol{x}, z, \theta)$ indicates parameter θ works well on example $(\boldsymbol{x}, z)$

- Given training data D, the **total loss** (or simply **loss function**) L is defined as

$$L(D, \theta) = \sum_{(\boldsymbol{x},z)\in D} \ell(\boldsymbol{x}, z, \theta)$$

➥ Objective = find "optimal" θ that **minimizes** $L(D, \theta)$

Learning NN = numerical optimization problem

- $\theta =$ parameters in NN (that we want to “optimize”)

  e.g., for two-layer feed-forward NNs, $\theta = \{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}\}$

- Define a **local loss function** $\ell(\boldsymbol{x}, z, \theta)$

  ➥ small $\ell(\boldsymbol{x}, z, \theta)$ indicates parameter θ works well on example $(\boldsymbol{x}, z)$

- Given training data D, the **total loss** (or simply **loss function**) L is defined as

$$L(D, \theta) = \sum_{(\boldsymbol{x}, z) \in D} \ell(\boldsymbol{x}, z, \theta)$$

➥ Objective = find “optimal” θ that **minimizes** $L(D, \theta)$

Learning NN = numerical optimization problem

- $\theta =$ parameters in NN (that we want to “optimize”)

  e.g., for two-layer feed-forward NNs, $\theta = \{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}\}$

- Define a **local loss function** $\ell(\boldsymbol{x}, z, \theta)$

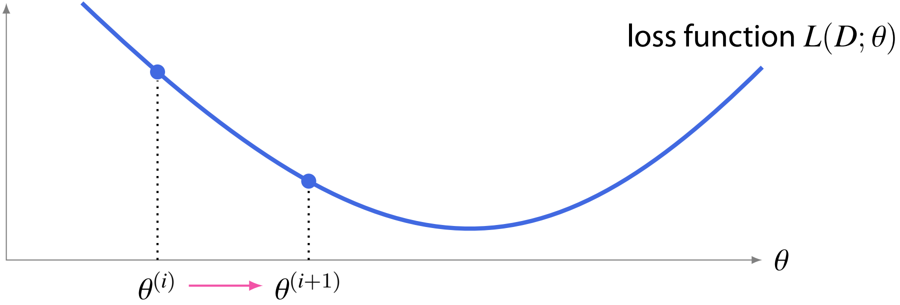
  ➥ small $\ell(\boldsymbol{x}, z, \theta)$ indicates parameter θ works well on example $(\boldsymbol{x}, z)$

- Given training data D, the **total loss** (or simply **loss function**) L is defined as

$$L(D, \theta) = \sum_{(\boldsymbol{x}, z) \in D} \ell(\boldsymbol{x}, z, \theta)$$

➥ Objective = find “optimal” θ that **minimizes** $L(D, \theta)$

Illustration: optimization



- (Total) loss function L is a function of θ (=parameters of neural networks)
- We search for the optimal θ that minimizes the loss
- Usually by a gradient-based method, such as **stochastic gradient descent (SGD)**

0/1-loss: intuitive, simple loss (but hard to optimize)

- Assume binary classification: $z \in \{-1, +1\}$
- 0/1-loss is then:

$$\ell_{0/1}(\boldsymbol{x}, z, \theta) = \begin{cases} 0 & \text{if } z \cdot NN(\boldsymbol{x}) \geq 0 \\ 1 & \text{otherwise} \end{cases}$$

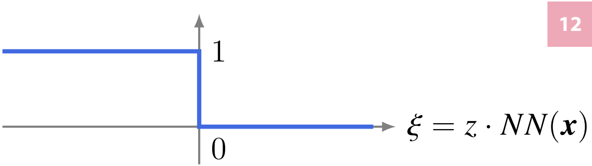
- We want to find θ that minimizes the total loss across training data:

$$\begin{aligned} L(D; \theta) &= \sum_i \ell_{0/1}(\boldsymbol{x}_i, z_i, \theta) \\ &= \text{(number of incorrectly classified training examples)} \end{aligned}$$

- We could argue that perceptron optimizes this loss (but perceptron is applicable only to linear classification)

Zero-one loss is difficult to optimize

$$\ell_{0/1}(\boldsymbol{x}, z, \theta) = \begin{cases} 0 & \text{if } z \cdot NN(\boldsymbol{x}) \geq 0 \\ 1 & \text{otherwise} \end{cases}$$



- Gradient with respect to $\xi = z \cdot NN(\boldsymbol{x})$ is zero everywhere except at $\xi = 0$
- At $\xi = 0$, ℓ is non-differentiable

➥ Gradient-based parameter optimization cannot be applied

∵ Current standard method for learning NNs, **stochastic gradient descent (SGD)**, requires the loss function to be continuous and differentiable

➥ We'll introduce some alternative loss functions later

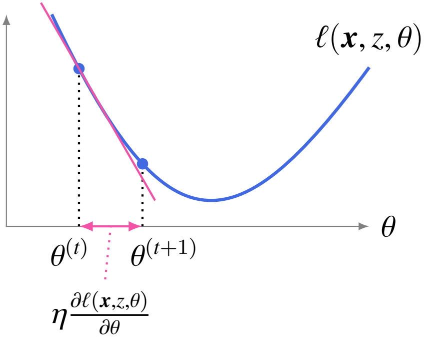
Gradient-based optimization

Intuition behind SGD:

Repeatedly take a small step in the direction that reduces the loss value

$$\theta \leftarrow \theta - \eta \frac{\partial \ell(\boldsymbol{x}, z, \theta)}{\partial \theta}$$

- Derivative of $\ell(\boldsymbol{x}, z, \theta)$ determines the direction (and also influences the step size) Note the negative sign—we are looking for a direction that reduces the loss)
- $\eta > 0$ determines the base step size, which must be set to a relatively small value



Outline

- Review on feed-forward neural networks
- Learning as optimization
- **Popular loss functions**
- Stochastic gradient descent (SGD)
- Back propagation

Hinge loss (also known as margin loss)

$$\ell_{\text{hinge}}(\boldsymbol{x}, z, \theta)$$
$$= \max(0, 1 - z \cdot NN(\boldsymbol{x}))$$

- Recall that classification is correct when $z \cdot NN(\boldsymbol{x}) \geq 0$
- Gradient is nonzero everywhere misclassification occurs ($z \cdot NN(\boldsymbol{x}) < 0$) but also $0 \leq z \cdot NN(\boldsymbol{x}) \leq 1$
- Loss becomes 0 only when $z \cdot NN(\boldsymbol{x}) \geq 1$
- ➥ Loss may be incurred even if classification is correct; i.e. when $0 \leq z \cdot NN(\boldsymbol{x}) \leq 1$
- ➥ Interpretation: penalize a classifier unless it can classify with a large confidence (=**margin**, which is 1 here)
- Not differentiable at $z \cdot NN(\boldsymbol{x}) = 1$, but “subderivative” can be used

Loss functions based on softmax

- Many other loss functions can be obtained by first transforming the output score $y_i = NN(\boldsymbol{x})$ to a probability, by softmax (last week)

$$\text{softmax}(y_i) = \frac{\exp(y_i)}{\sum_{j \in \mathcal{Y}} \exp(y_j)}$$

  where $\mathcal{Y} = \{1, 2, \ldots, K\}$ is the set of classes
- We can then define several differentiable losses from that probability

  Cross-entropy loss, etc.

Softmax for binary classification 1/2

- Recall the output of softmax for multi-class classification is:

$$\text{softmax}(y_i) = \frac{\exp(y_i)}{\sum_{j \in \mathcal{Y}} \exp(y_j)}$$

- For binary-classification, output is a single value (a scalar) $y = NN(\boldsymbol{x})$, so we cannot use this formula
- The softmax for binary classification is defined as:

$$p(z|\boldsymbol{x}) = \text{softmax}(NN(\boldsymbol{x})) = \frac{1}{1 + \exp(-2z \cdot NN(\boldsymbol{x}))}$$

for $z \in \mathcal{Y} = \{-1, +1\}$

Softmax for binary classification 2/2

$$p(z|\boldsymbol{x}) = \text{softmax}(NN(\boldsymbol{x})) = \frac{1}{1 + \exp(-2z \cdot NN(\boldsymbol{x}))} \tag{1}$$

To see why,

- The (unnormalized) score to select $z \in \{-1, +1\}$ is $\exp(z \cdot NN(\boldsymbol{x}))$.
- Thus, the normalized score (probability) is:

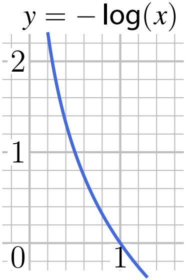
$$p(z|\boldsymbol{x}) = \frac{\exp(z \cdot NN(\boldsymbol{x}))}{\exp(+1 \cdot NN(\boldsymbol{x})) + \exp(-1 \cdot NN(\boldsymbol{x}))}$$

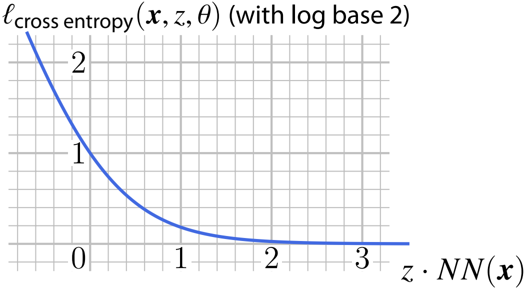
- Multiplying both numerator and denominator by $\exp(-z \cdot NN(\boldsymbol{x}))$ yields Eq. (1) (for both $z = -1$ and $z = +1$)

Cross entropy loss (also called log loss)

$$\ell_{\text{cross entropy}}(\boldsymbol{x}, z, \theta) = -\log p(z|\boldsymbol{x}) = -\log \frac{1}{1 + \exp(-2z \cdot NN(\boldsymbol{x}))}$$

Cross entropy loss decreases as the probability of choosing the correct label (i.e., $p(z|\boldsymbol{x})$ with correct label z) approaches 1 (in which case $\ell_{\text{cross entropy}} \to 0$)





Multi-class cross-entropy loss

- For multi-class classification (over set of classes $\mathcal{Y} = \{1, \ldots, K\}$),

$$p(z|\boldsymbol{x}) = \text{softmax}(y_z) = \frac{\exp(y_z)}{\sum_{j \in \mathcal{Y}} \exp(y_j)}$$

- With this “probability”, **cross entropy** is defined as:

$$\ell_{\text{cross entropy}}(\boldsymbol{x}, z, \theta) = -\log p(z|\boldsymbol{x})$$

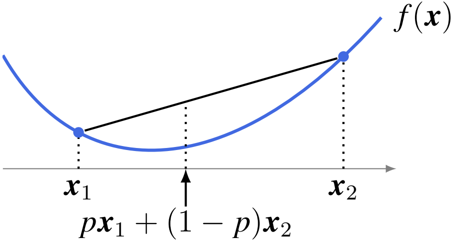
One of the most popular loss functions for training NNs these days

Convexity of loss function

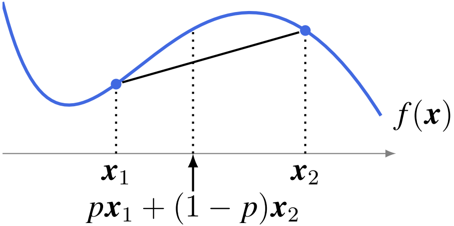
Definition $f(\boldsymbol{x})$ is **convex** if for any $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathcal{X}$ and $0 < p < 1$:

$$f(p\boldsymbol{x}_1 + (1-p)\boldsymbol{x}_2) \leq pf(\boldsymbol{x}_1) + (1-p)f(\boldsymbol{x}_2)$$

convex



non-convex

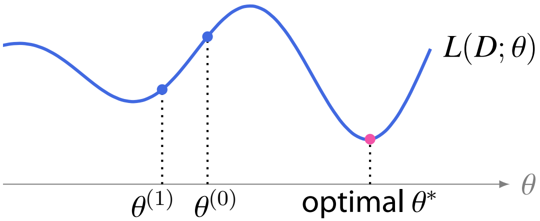


Some operations of two functions preserve convexity

- e.g., sum or sup ($\simeq$ max) of two convex functions is convex
- but **composition** of two convex functions is **not** convex in general

Optimizing neural networks is a non-convex problem

- Because loss L is a composition of several functions, it is usually non-convex
- No guarantee that SGD (and other gradient-based methods) find the global optimum (=the best point minimizing L) when L is non-convex
- The minimum found by SGD is merely a **local** optimum

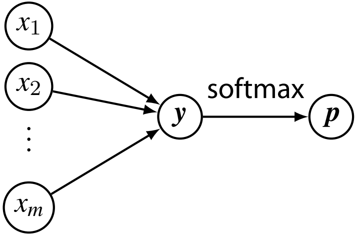


Starting from $\theta = \theta^{(0)}$, SGD will move θ to the left, because that is the direction where L is decreased (around $\theta^{(0)}$) (although optimal value θ^* lies on the right)

Logistic regression is convex

If the network does not contain hidden layers, and the loss is given by cross entropy, it is called **logistic regression**:

$$\boldsymbol{p} = \text{softmax}(\boldsymbol{y}) = \text{softmax}(\overbrace{\boldsymbol{W}\boldsymbol{x}}^{\boldsymbol{y}})$$



- The loss function of logistic regression is convex
  - ➥ the global optimum can be found with SGD
- For many years, learning NN (= non-convex optimization) was thought to be impractical; logistic regression was popular thanks to its convexity
  - ➥ Although NNs have no such guarantee, NNs (with local minima found by SGD) is often more effective than logistic regression

Outline

- Review on feed-forward neural networks
- Learning as optimization
- Popular loss functions
- **Stochastic gradient descent (SGD)**
- Back propagation

Stochastic gradient descent (SGD)

$$D = \{(\boldsymbol{x}_1, z_1), (\boldsymbol{x}_2, z_2), \cdots, (\boldsymbol{x}_N, z_N)\}$$

$$L(D; \theta) = \sum_{i=1}^{N} \ell(\boldsymbol{x}_i, z_i, \theta)$$

- Our goal is to minimize the total loss $L(D; \theta)$. Natural update formula would be

  $$\theta \leftarrow \theta - \eta \frac{\partial L(D, \theta)}{\partial \theta}$$

  However, $\partial L / \partial \theta$ is a computational burden (both in terms of speed and memory)
- SGD instead looks at only one example $(\boldsymbol{x}_i, z_i)$ at a time, and take the derivative of the local loss $\ell(\boldsymbol{x}_i, z_i, \theta)$

  $$\theta \leftarrow \theta - \eta \frac{\partial \ell(\boldsymbol{x}_i, z_i, \theta)}{\partial \theta}$$

➥ A “perceptron-like”learning algorithm

More about derivative

Recall that θ is a collection of parameters; e.g., $\theta = \{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \cdots\}$

- θ can be seen as a (huge) vector $\boldsymbol{\theta} = (w_{11}^{(1)}, w_{12}^{(1)}, \cdots, w_{11}^{(2)}, w_{12}^{(2)}, \cdots)^{\mathsf{T}}$

$$\theta \leftarrow \theta - \eta \frac{\partial \ell(\boldsymbol{x}_i, z_i, \theta)}{\partial \theta}$$

scalar

vector

- A derivative of a scalar with respect to a vector is also a vector

$$\frac{\partial \ell(\boldsymbol{x}_i, z_i, \theta)}{\partial \theta} = \begin{pmatrix} \frac{\partial \ell}{w_{11}^{(1)}} \\ \frac{\partial \ell}{w_{12}^{(1)}} \\ \vdots \end{pmatrix}$$

- The central problem is then how to obtain each derivative

- **Back-propagation** is a general solution (later)

SGD algorithm

```
Initialize $\theta$
**repeat**
    Randomly pick up a training example $(\boldsymbol{x}, z) \in D$
    Compute the loss $\ell(\boldsymbol{x}, z, \theta)$
    Update: $\theta \leftarrow \theta - \eta \frac{\partial \ell(\boldsymbol{x}, z, \theta)}{\partial \theta}$
**until** “convergence”
```

There are several possible criteria for convergence, e.g.,

- loss does not decrease (or the change is sufficiently small)
- performance (or loss) on **validation (development) data** does not improve

What is validation (development) data?

- Validation data = additional labeled data, with no overlap with training data
- We could split the available labeled data into 80 (for training) : 20 (for development)
- In SGD, judging convergence using only training data often leads to **overfitting** (= model works perfectly on training data, but fails on new unseen data)
- To avoid overfitting, convergence is detected in terms of loss or the accuracy on the validation data

Tricks to escape from local optimum

- Recall: NN loss is non-convex; the parameter found is generally not the global optimum
- There are several tricks to find a better local optimum (achieving a smaller loss); examples are:
    - Initialization
    - Adjusting learning rate

Initialization

- Often, each weight is randomly initialized on a small range
- Consider a weight matrix $\boldsymbol{W} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$
- Two known techniques (available in most NN libraries):

Sample from a normal (Gaussian) distribution [He et al., 2015]

Works well for ReLU activation function

$$w_{ij} \leftarrow \text{Normal}(0, \sqrt{2/d_{\text{in}}})$$

Xavier initialization [Glorot and Bengio, 2010]

Suitable for tanh, etc.

$$w_{ij} \leftarrow \text{Uniform}\left[-\frac{\sqrt{6}}{\sqrt{d_{\text{in}} + d_{\text{out}}}}, +\frac{\sqrt{6}}{\sqrt{d_{\text{in}} + d_{\text{out}}}}\right]$$

Learning rate

- A constant learning rate η often does not perform well
- Even when using a constant rate, we have to select its value
- Usually we try several values in $[0, 1]$; e.g., $0.001, 0.01, 0.1, 1$ (and choose the one that performs best on validation data)
- A popular approach for SGD is to decrease the learning rate gradually for each update [Button, 2012]:

$$\eta_t = \eta_0(1 + \eta_0\lambda t)^{-1}$$

where

  - t: number of updates carried out since the start of training
  - η_0: the initial learning rate
  - λ: a hyper parameter (must be set by the user)

Beyond SGD

- SGD with a constant rate is still a competitive method, but recently several alternatives methods have been proposed—these adaptively adjust the learning rate
  - Momentum [Rumelhard et al., 1986]
  - AdaGrad [Duchi et al., 2011]
  - Adam [Kingma and Ba., 2014]
- Adam has been popular these days, but is still not perfect
- The following is a good survey for optimizers:

Sebastian Ruder.
An overview of gradient descent optimization algorithms.
arXiv preprint arXiv:1609.04747, 2016.

Mini-batch training

- Instead of using a single example for each update, mini-batch training calculates an **accumulated gradient** for data subset D_i
- First divide the training data D into subsets $(D_1, D_2, \cdots, D_n)$
- Each D_i contains typically 10–100 training example; then

$$\theta \leftarrow \theta - \eta \frac{1}{|D_i|} \sum_{(\boldsymbol{x}_j, z_j) \in D_i} \frac{\partial \ell(\boldsymbol{x}_j, z_j, \theta)}{\partial \theta}$$

- Advantages:
    - computationally efficient (especially for GPU) because we can pack several matrix and vector multiplications into one matrix and matrix multiplication
    - Learning is stabilized as several examples are optimized simultaneously

Other techniques for improved learning

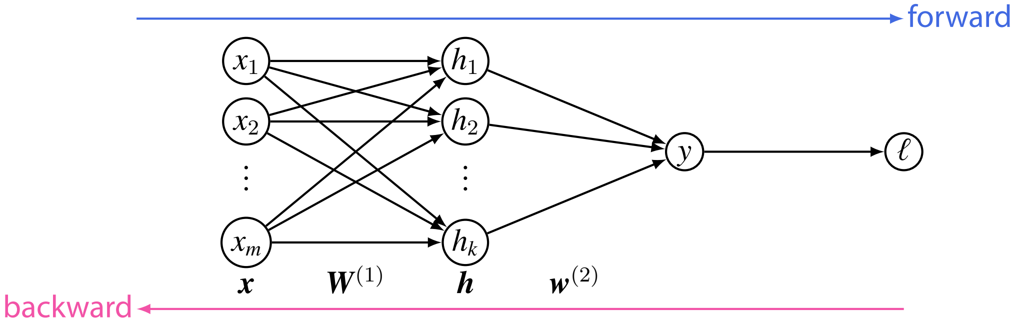
- Regularization
  - A popular (and classical) way to prevent overfitting.
  - Add to the loss a term: $\|\theta\|^2 \; (\|\boldsymbol{w}\|^2 = w_1^2 + w_2^2 + \cdots)$
    Each value of θ is encouraged to be small
  - Prevents a small number of variables to be too large
- Dropout
  - One of the key techniques for the recent success of deep learning
  - The model tries to classify with only a subset of parameters
  - During training, we randomly select one half of nodes and ignore them

Outline

- Review on feed-forward neural networks
- Learning as optimization
- Popular loss functions
- Stochastic gradient descent (SGD)
- **Back propagation**

Back propagation

- A generic technique to compute the value of $\partial \ell / \partial \theta$ at the current parameter θ
- What's the meaning of "back" in back propagation?
  - "Forward" = calculating loss as a function of the input and the parameters
  - "Backward" = calculating the derivative with respect to each parameter—starting from the output (loss) and traversing backward in the network, using the values computed in the forward pass on the way



Example neural network

Let the non-linear activation funciton be ReLU, and the loss be hinge loss

$$
\begin{aligned}
\boldsymbol{a} &= \boldsymbol{W}^{(1)}\boldsymbol{x} \\
\boldsymbol{h} &= \text{ReLU}(\boldsymbol{a}) \\
y &= \boldsymbol{w}^{(2)} \cdot \boldsymbol{h} \\
\ell &= \text{max}(0, 1 - yz)
\end{aligned}
$$

For SGD update

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\partial \ell(\boldsymbol{x}_i, z_i, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$$

we need to calculate $\partial \ell / \partial \boldsymbol{\theta}$

Specifically, we need $\frac{\partial \ell}{\partial \boldsymbol{W}^{(1)}}$ and $\frac{\partial \ell}{\partial \boldsymbol{w}^{(2)}}$ (because $\theta = \{\boldsymbol{W}^{(1)}, \boldsymbol{w}^{(2)}\}$ here)

Back propagation: Use "chain rule"

- Observe that ℓ is a function of y, and y is a function of $\boldsymbol{w}^{(2)}$

➥ We can apply **chain rules** to obtain derivatives

$$\frac{\partial \ell}{\partial \boldsymbol{w}^{(2)}} = \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial \boldsymbol{w}^{(2)}}$$

$$\boldsymbol{a} = \boldsymbol{W}^{(1)}\boldsymbol{x}$$

$$\boldsymbol{h} = \text{ReLU}(\boldsymbol{a})$$

$$y = \boldsymbol{w}^{(2)} \cdot \boldsymbol{h}$$

$$\ell = \max(0, 1 - yz)$$

- Similarly,

$$\frac{\partial \ell}{\partial \boldsymbol{W}^{(1)}} = \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{W}^{(1)}}$$

Point: Traverse the network backward from ℓ to the target parameter, and then connect their partial derivatives with the chain rule

Backpropagation: Hand calculation

$$\ell = \mathsf{max}(0, 1 - yz)$$

$$y = \boldsymbol{w}^{(2)} \cdot \boldsymbol{h}$$

$$\frac{\partial \ell}{\partial y} = \begin{cases} 0 & yz \geq 1 \\ -z & yz < 1 \end{cases}$$

$$\frac{\partial y}{\partial \boldsymbol{w}^{(2)}} = \boldsymbol{h}$$

$$\frac{\partial \ell}{\partial \boldsymbol{w}^{(2)}} = \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial \boldsymbol{w}^{(2)}} \left(= \begin{cases} 0 & yz \geq 1 \\ -z\boldsymbol{h} & yz < 1 \end{cases} \right)$$

- Values y, z, and $\boldsymbol{h}$ are all available as the result of **forward** computation

➥ $\partial \ell / \partial y$, $\partial y / \partial \boldsymbol{w}^{(2)}$, and hence $\partial \ell / \partial \boldsymbol{w}^{(2)}$ are all computable by the formula above

- We thus obtained $\partial\ell/\boldsymbol{w}^{(2)}$.
- We can do a similar backpropagation computation for $\partial\ell/\boldsymbol{W}^{(1)}$ using chain rules (quite involved, and hence omitted).
- “Vectorizing” the components of these two derivatives, we obtain $\partial\ell/\partial\boldsymbol{\theta}$.

Recall, in two-layer feed-forward neural network (of this example)

$$\theta = \{\boldsymbol{W}^{(1)}, \boldsymbol{w}^{(2)}\}$$

are the only parameters.

We can thus apply SGD update:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\frac{\partial\ell(\boldsymbol{x}_i, z_i, \boldsymbol{\theta})}{\partial\boldsymbol{\theta}}$$

Notes

- However, hand calculation of backpropagation is an error prone process for more complex networks

  Note that even the calculation of $\frac{\partial \ell}{\partial \boldsymbol{W}^{(1)}}$ in our previous small network is quite involved (and thus was omitted))

- Fortunately, this calculation can be automated
- Useful tools: **computational graph** and **automatic differentiation**
  ➥ see e.g., Goodfellow, Bengio, Courville, "Deep Learning" MIT Press, Sec. 6.5

Summary

Loss-based learning

define a loss function, and learning the parameters to reduce the loss on the training data

Convexity

Global optimum can be found if the loss function is convex; this is not true for NNs; true for logistic regression

SGD

An online gradient-based method for finding local optimum

Back propagation

Calculate gradients with respect to parameters using the chain rule