Mnacho Echenim
small evolutions : Patrick Reignier

Grenoble INP-Ensimag

2025-2026

Outline

A beginner's trap

- It is standard to initialize all biases to 0

A beginner's trap

- It is standard to initialize all biases to 0
- What if we initialize all weights to 0?
  - More generally, what if we initialize all weights at layer j to the same value σ^j?

A beginner's trap

- It is standard to initialize all biases to 0
- What if we initialize all weights to 0?
  - More generally, what if we initialize all weights at layer j to the same value ω^j?
- Forward propagation: for all layers j, net input and activation components have the same value: $\forall k, p \in \{1 \ldots, n_j\}$, $\zeta_k^j = \zeta_p^j$, hence $\alpha_k^j = \alpha_p^j$ (induction)

A beginner's trap

- It is standard to initialize all biases to 0
- What if we initialize all weights to 0?
  - More generally, what if we initialize all weights at layer j to the same value $\mathfrak{w}^j$?
- Forward propagation: for all layers j, net input and activation components have the same value: $\forall k, p \in \{1 \dots, n_j\}$, $\zeta_k^j = \zeta_p^j$, hence $\alpha_k^j = \alpha_p^j$ (induction)
- Backpropagation: for all layers j, error components have the same value: $\forall k, p \in \{1 \dots, n_j\}$, $\mathcal{B}_k^j = \mathcal{B}_p^j = \mathfrak{b}^j$ (backward induction)

A beginner's trap

- It is standard to initialize all biases to 0
- What if we initialize all weights to 0?
  - More generally, what if we initialize all weights at layer j to the same value $\mathfrak{o}^j$?
- Forward propagation: for all layers j, net input and activation components have the same value: $\forall k, p \in \{1 \dots, n_j\}$, $\zeta_k^j = \zeta_p^j$, hence $\alpha_k^j = \alpha_p^j$ (induction)
- Backpropagation: for all layers j, error components have the same value: $\forall k, p \in \{1 \dots, n_j\}$, $\mathcal{B}_k^j = \mathcal{B}_p^j = \mathfrak{b}^j$ (backward induction)
- Gradient computation: for all layers j we have

$$\nabla_{W^j} \mathcal{C} \;=\; \alpha^{j-1} \cdot \left[\mathcal{B}^j\right]^{\mathrm{T}} \;=\; \alpha^{j-1} \cdot (\mathfrak{b}^j, \dots, \mathfrak{b}^j),$$

  so that the weight upgrade for each neuron in layer j is $\mathfrak{b}^j \cdot \alpha^{j-1}$

A beginner's trap

- It is standard to initialize all biases to 0
- What if we initialize all weights to 0?
  - More generally, what if we initialize all weights at layer j to the same value $\mathfrak{o}^j$?
- Forward propagation: for all layers j, net input and activation components have the same value: $\forall k, p \in \{1 \dots, n_j\}$, $\zeta_k^j = \zeta_p^j$, hence $\alpha_k^j = \alpha_p^j$ (induction)
- Backpropagation: for all layers j, error components have the same value: $\forall k, p \in \{1 \dots, n_j\}$, $\mathcal{B}_k^j = \mathcal{B}_p^j = \mathfrak{b}^j$ (backward induction)
- Gradient computation: for all layers j we have

$$\nabla_{W^j} \mathcal{C} \;=\; \alpha^{j-1} \cdot \left[\mathcal{B}^j\right]^{\mathrm{T}} \;=\; \alpha^{j-1} \cdot (\mathfrak{b}^j, \dots, \mathfrak{b}^j),$$

  so that the weight upgrade for each neuron in layer j is $\mathfrak{b}^j \cdot \alpha^{j-1}$

Conclusion

The weights of neurons in a same layer are updated the same way, and the network cannot be trained efficiently

Initial weights

- The network obtained after the training phase can strongly depend on the initial weights

Initial weights

- The network obtained after the training phase can strongly depend on the initial weights
  - If the initial network is at a local minimum, training will be useless

Initial weights

- The network obtained after the training phase can strongly depend on the initial weights
  - If the initial network is at a local minimum, training will be useless
  - If all initial weights in a layer are equal, they will be updated in the same way

Initial weights

- The network obtained after the training phase can strongly depend on the initial weights
  - If the initial network is at a local minimum, training will be useless
  - If all initial weights in a layer are equal, they will be updated in the same way
- Standard technique: random initialization
  - But depends on the architecture of the network

Initial weights

- The network obtained after the training phase can strongly depend on the initial weights
  - If the initial network is at a local minimum, training will be useless
  - If all initial weights in a layer are equal, they will be updated in the same way
- Standard technique: random initialization
  - But depends on the architecture of the network
  - Gaussian initialization: weights of layer i are initialized with a value drawn from a Gaussian distribution with 0 mean and $1/\sqrt{n_{i-1}}$ standard deviation, where n_{i-1} is the size of layer $i-1$

Initial weights

- The network obtained after the training phase can strongly depend on the initial weights
  - If the initial network is at a local minimum, training will be useless
  - If all initial weights in a layer are equal, they will be updated in the same way
- Standard technique: random initialization
  - But depends on the architecture of the network
  - Gaussian initialization: weights of layer i are initialized with a value drawn from a Gaussian distribution with 0 mean and $1/\sqrt{n_{i-1}}$ standard deviation, where n_{i-1} is the size of layer $i-1$
  - Xavier initialization: weights of layer i are initialized with a value drawn from the Uniform distribution on $\left[-\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}\right]$

Initial weights

- The network obtained after the training phase can strongly depend on the initial weights
  - If the initial network is at a local minimum, training will be useless
  - If all initial weights in a layer are equal, they will be updated in the same way
- Standard technique: random initialization
  - But depends on the architecture of the network
  - Gaussian initialization: weights of layer i are initialized with a value drawn from a Gaussian distribution with 0 mean and $1/\sqrt{n_{i-1}}$ standard deviation, where n_{i-1} is the size of layer $i-1$
  - Xavier initialization: weights of layer i are initialized with a value drawn from the Uniform distribution on $\left[-\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}\right]$

Note

- Initialization schemes frequently depend on the activation function that is used
- For example, biases are sometimes initialized to a small value when using ReLU activations

Outline

Activation functions

- Theoretically, any nonpolynomial (derivable) function could be used

Activation functions

- Theoretically, any nonpolynomial (derivable) function could be used
- Issues to consider:

Activation functions

- Theoretically, any nonpolynomial (derivable) function could be used
- Issues to consider:
  - Computation cost of applying the function

Activation functions

- Theoretically, any nonpolynomial (derivable) function could be used
- Issues to consider:
  - Computation cost of applying the function
  - Computation cost of applying the derivative of the function

Activation functions

- Theoretically, any nonpolynomial (derivable) function could be used
- Issues to consider:
  - Computation cost of applying the function
  - Computation cost of applying the derivative of the function
  - The issue of vanishing/exploding gradients

Activation functions

- Theoretically, any nonpolynomial (derivable) function could be used
- Issues to consider:
  - Computation cost of applying the function
  - Computation cost of applying the derivative of the function
  - The issue of vanishing/exploding gradients
  - The issue of dead neurons

Activation functions

- Theoretically, any nonpolynomial (derivable) function could be used
- Issues to consider:
  - Computation cost of applying the function
  - Computation cost of applying the derivative of the function
  - The issue of vanishing/exploding gradients
  - The issue of dead neurons
- See also Stanford lecture:
  `https://www.youtube.com/watch?v=wEoyxE0GP2M`

Sigmoid (logistic) activation function

- $\sigma(x) = \frac{1}{1+\exp(-x)}$

Sigmoid (logistic) activation function

- $\sigma(x) = \frac{1}{1+\exp(-x)}$
- $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$

Sigmoid (logistic) activation function

- $\sigma(x) = \frac{1}{1+\exp(-x)}$
- $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$
- Used to be popular: biological interpretation

Sigmoid (logistic) activation function

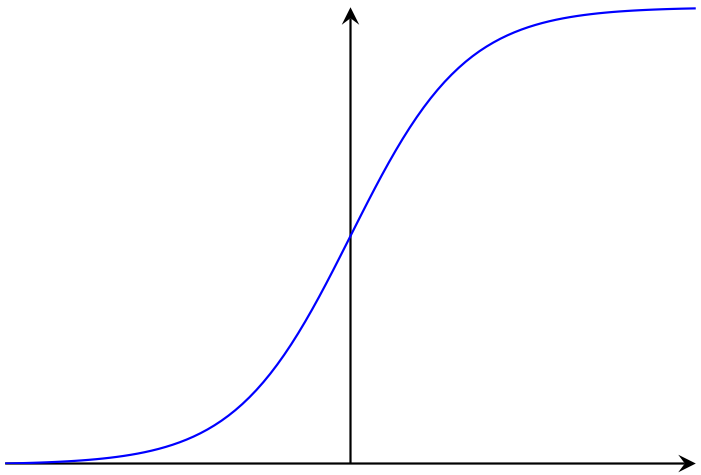
- $\sigma(x) = \frac{1}{1+\exp(-x)}$
- $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$
- Used to be popular: biological interpretation
- Output is always strictly positive

Sigmoid (logistic) activation function

- $\sigma(x) = \frac{1}{1+\exp(-x)}$
- $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$
- Used to be popular: biological interpretation
- Output is always strictly positive
- Saturated neurons produce 0 gradient

Sigmoid (logistic) activation function

- $\sigma(x) = \frac{1}{1+\exp(-x)}$
- $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$
- Used to be popular: biological interpretation
- Output is always strictly positive
- Saturated neurons produce 0 gradient
- $\sigma'(x) \leq 0.25 \Longrightarrow$ vanishing gradient problem



The problem with positive inputs

- Assume α^j, the input to layer $j+1$, is always positive

The problem with positive inputs

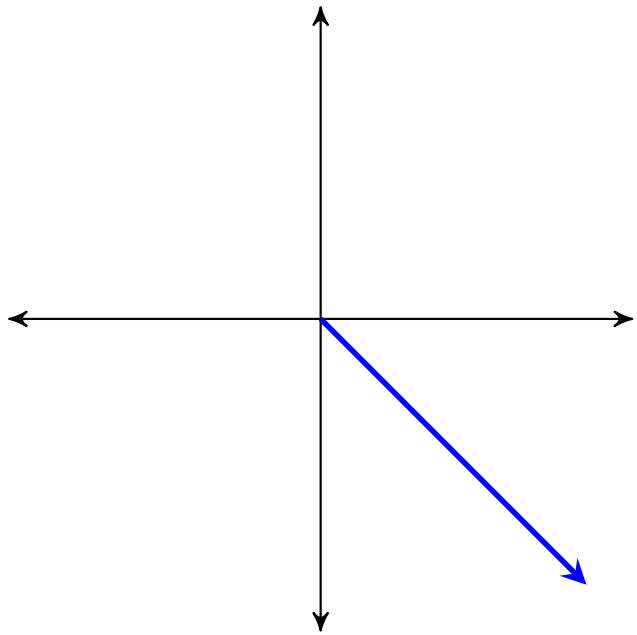
- Assume α^j, the input to layer $j+1$, is always positive
- Recall that the weight vector of neuron ν_k^{j+1} is updated by $\mathcal{B}_k^{j+1} \cdot \alpha^j$

The problem with positive inputs

- Assume α^j, the input to layer $j+1$, is always positive
- Recall that the weight vector of neuron ν_k^{j+1} is updated by $\mathcal{B}_k^{j+1} \cdot \alpha^j$
- Thus, all its components are updated in the same direction, depending on the sign of $\mathcal{B}_k^{j+1}$

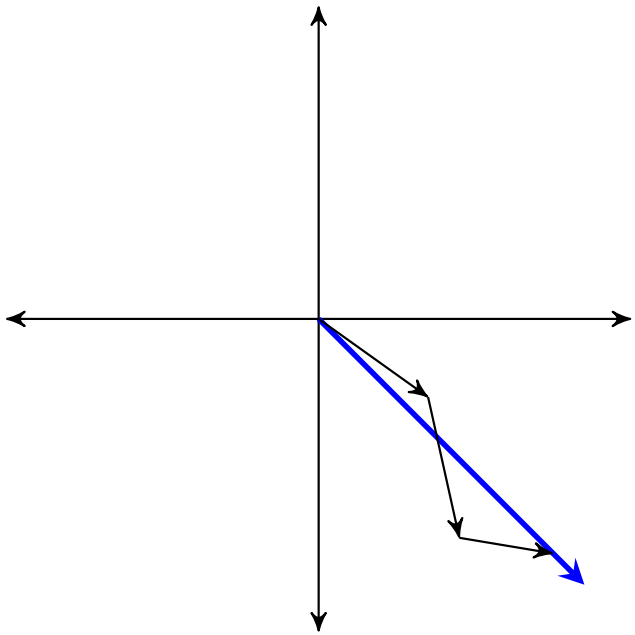
The problem with positive inputs

- Assume α^j, the input to layer $j+1$, is always positive
- Recall that the weight vector of neuron ν_k^{j+1} is updated by $\mathcal{B}_k^{j+1} \cdot \alpha^j$
- Thus, all its components are updated in the same direction, depending on the sign of $\mathcal{B}_k^{j+1}$



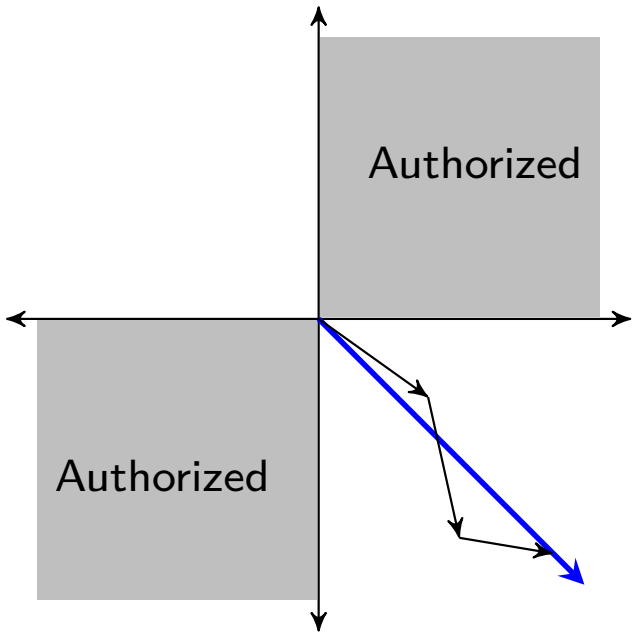
The problem with positive inputs

- Assume α^j, the input to layer $j+1$, is always positive
- Recall that the weight vector of neuron ν_k^{j+1} is updated by $\mathcal{B}_k^{j+1} \cdot \alpha^j$
- Thus, all its components are updated in the same direction, depending on the sign of $\mathcal{B}_k^{j+1}$



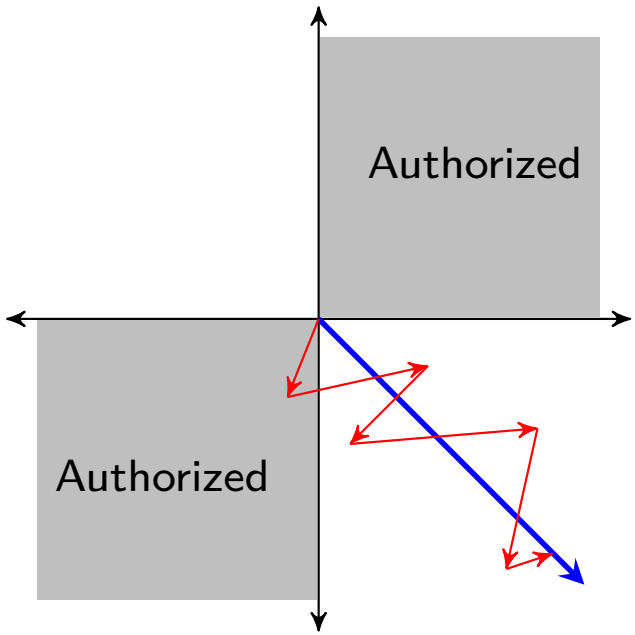
The problem with positive inputs

- Assume α^j, the input to layer $j+1$, is always positive
- Recall that the weight vector of neuron ν_k^{j+1} is updated by $\mathcal{B}_k^{j+1} \cdot \alpha^j$
- Thus, all its components are updated in the same direction, depending on the sign of $\mathcal{B}_k^{j+1}$

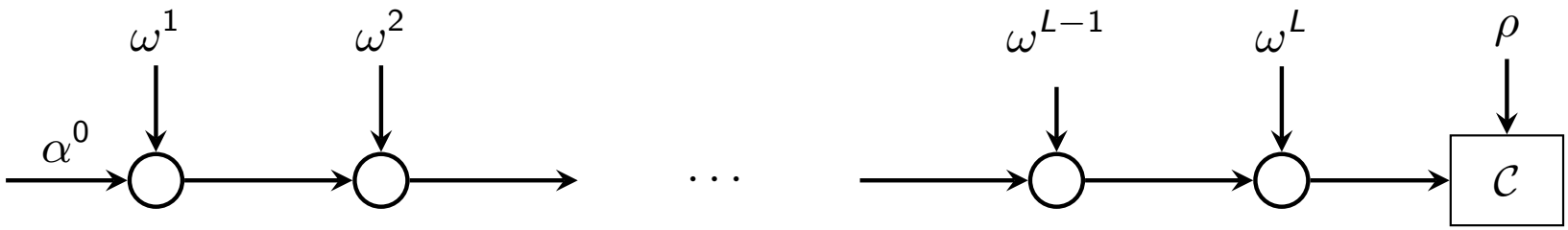


The problem with positive inputs

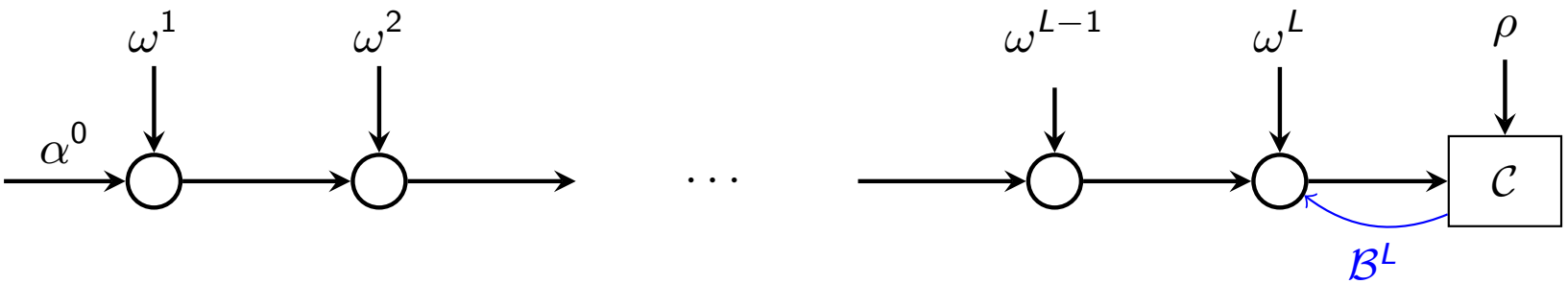
- Assume α^j, the input to layer $j+1$, is always positive
- Recall that the weight vector of neuron ν_k^{j+1} is updated by $\mathcal{B}_k^{j+1} \cdot \alpha^j$
- Thus, all its components are updated in the same direction, depending on the sign of $\mathcal{B}_k^{j+1}$



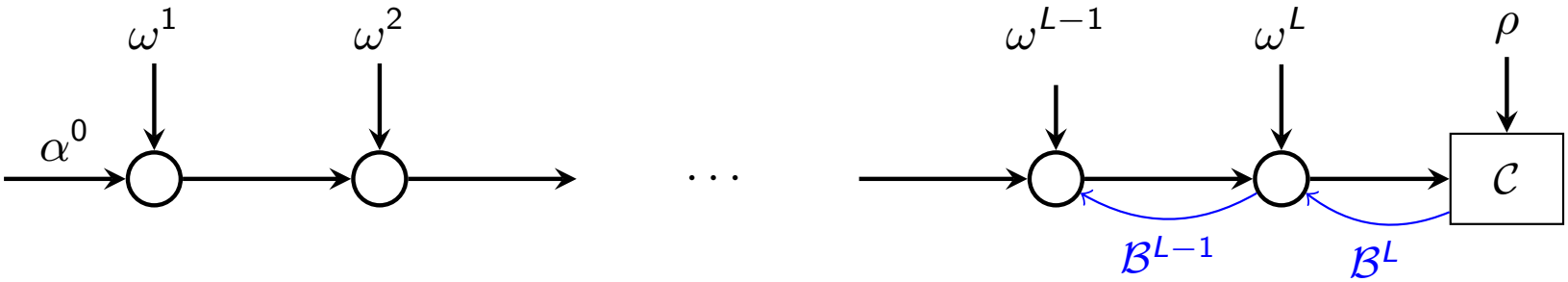
Vanishing gradients: illustration



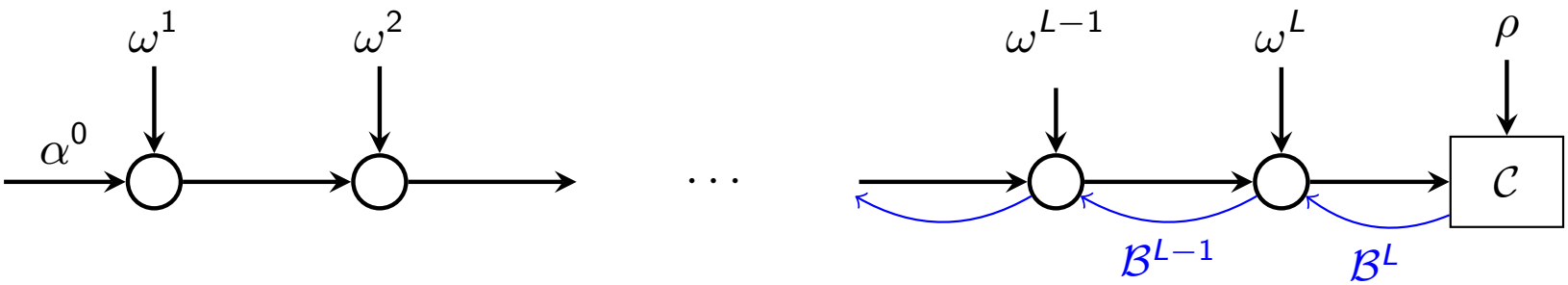
Vanishing gradients: illustration



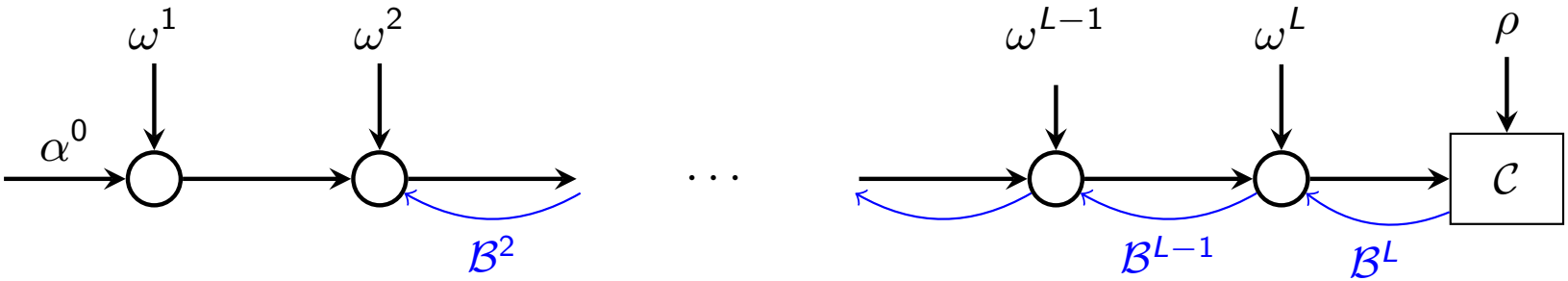
Vanishing gradients: illustration



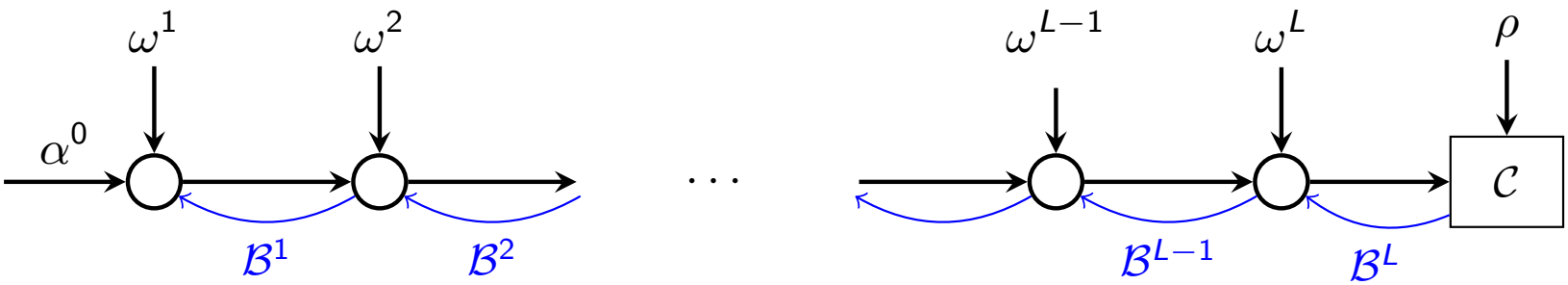
Vanishing gradients: illustration



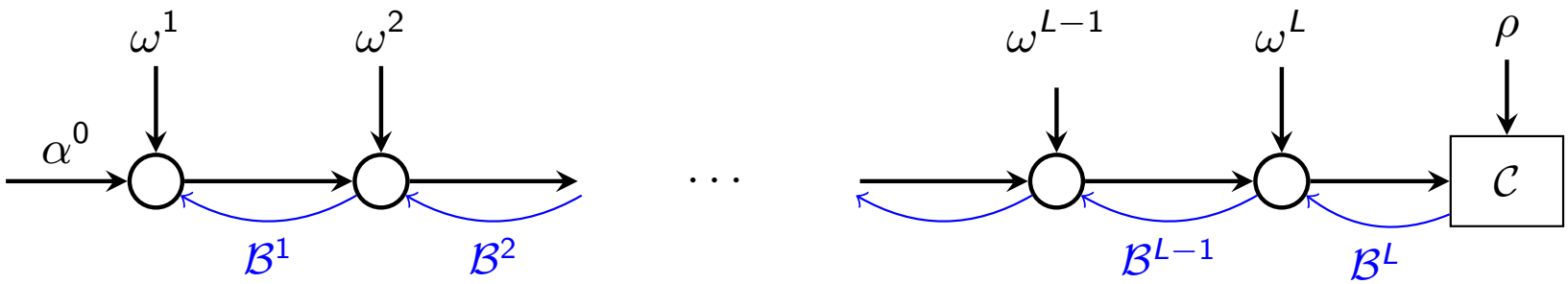
Vanishing gradients: illustration



Vanishing gradients: illustration



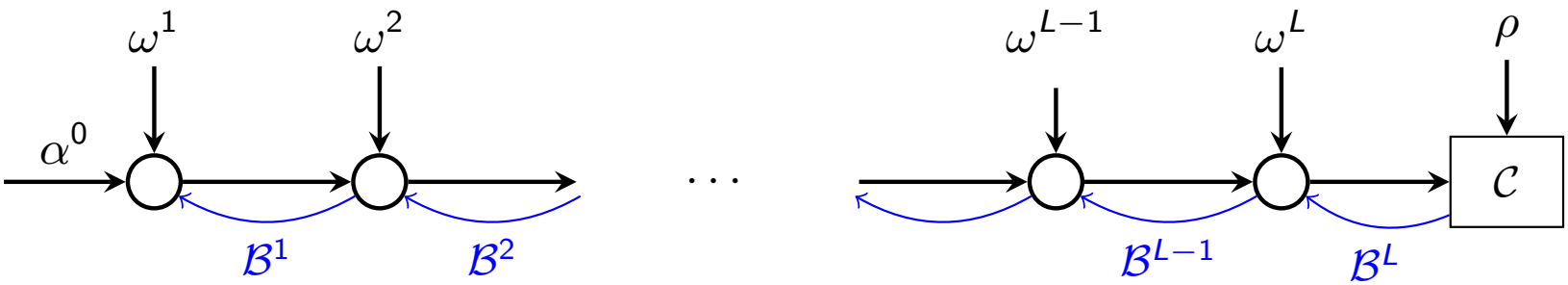
Vanishing gradients: illustration



We have:

$$\mathcal{B}^L = \Phi'(\zeta^L) \cdot \mathcal{C}'(\alpha^L, \rho) \leq \frac{\mathcal{C}'(\alpha^L, \rho)}{4}$$

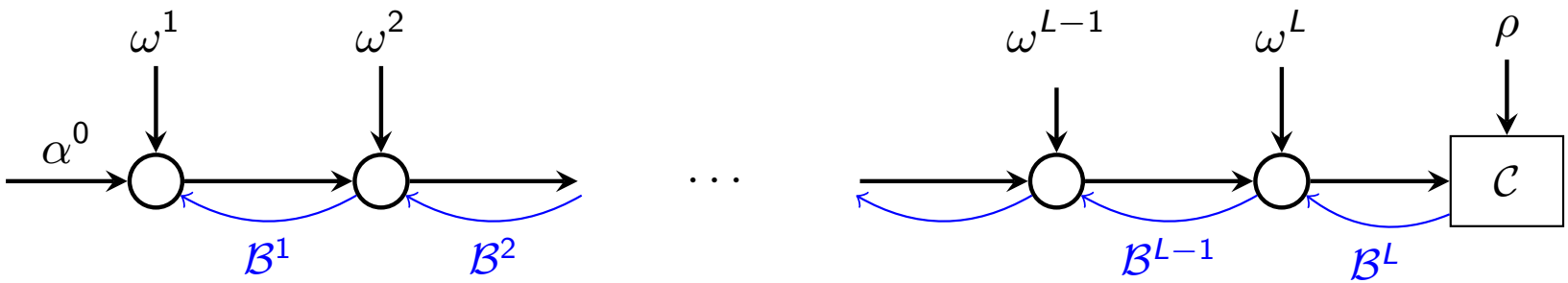
Vanishing gradients: illustration



We have:

$$\begin{aligned} \mathcal{B}^L &= \Phi'(\zeta^L) \cdot \mathcal{C}'(\alpha^L, \rho) &\leq \frac{\mathcal{C}'(\alpha^L,\rho)}{4} \\ \mathcal{B}^{L-1} &= \Phi'(\zeta^{L-1}) \cdot \omega^L \cdot \mathcal{B}^L &\leq \omega^L \cdot \frac{\mathcal{C}'(\alpha^L,\rho)}{4^2} \end{aligned}$$

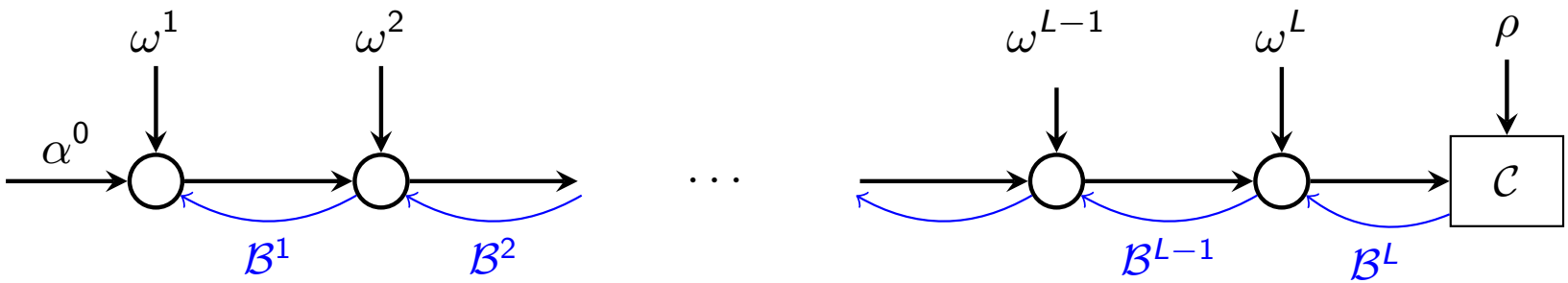
Vanishing gradients: illustration



We have:

$$\begin{aligned}
\mathcal{B}^L &= \Phi'(\zeta^L)\cdot\mathcal{C}'(\alpha^L,\rho) &&\leq \frac{\mathcal{C}'(\alpha^L,\rho)}{4}\\
\mathcal{B}^{L-1} &= \Phi'(\zeta^{L-1})\cdot\omega^L\cdot\mathcal{B}^L &&\leq \omega^L\cdot\frac{\mathcal{C}'(\alpha^L,\rho)}{4^2}\\
&\vdots\\
\mathcal{B}^i &= \Phi'(\zeta^i)\cdot\omega^{i+1}\cdot\mathcal{B}^{i+1} &&\leq \left(\prod_{j=L}^{i+1}\omega^j\right)\cdot\frac{\mathcal{C}'(\alpha^L,\rho)}{4^{L-i+1}}
\end{aligned}$$

Vanishing gradients: illustration

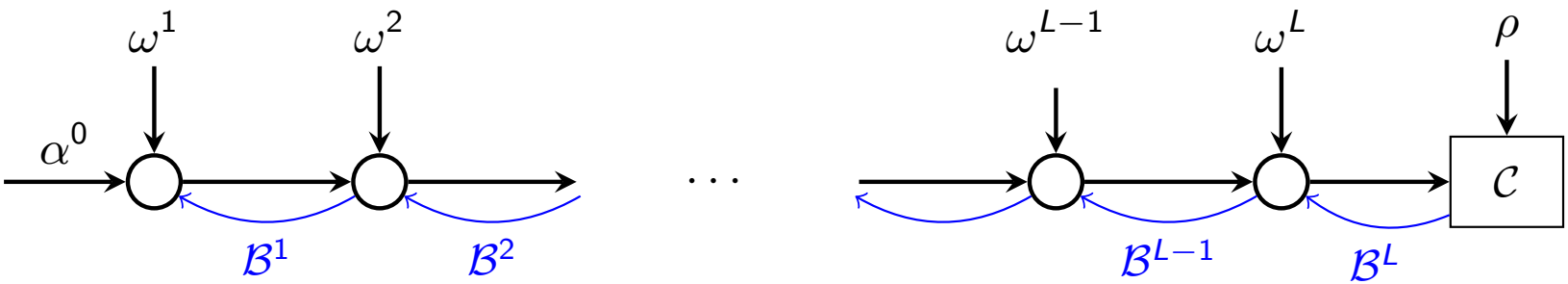


We have:

$$\begin{array}{rcll} \mathcal{B}^L & = & \Phi'(\zeta^L)\cdot\mathcal{C}'(\alpha^L,\rho) & \leq \frac{\mathcal{C}'(\alpha^L,\rho)}{4} \\ \mathcal{B}^{L-1} & = & \Phi'(\zeta^{L-1})\cdot\omega^L\cdot\mathcal{B}^L & \leq \omega^L\cdot\frac{\mathcal{C}'(\alpha^L,\rho)}{4^2} \\ & \vdots & & \\ \mathcal{B}^i & = & \Phi'(\zeta^i)\cdot\omega^{i+1}\cdot\mathcal{B}^{i+1} & \leq \left(\prod_{j=L}^{i+1}\omega^j\right)\cdot\frac{\mathcal{C}'(\alpha^L,\rho)}{4^{L-i+1}} \end{array}$$

- Except in the case where the weights are large, the 4^{L-i+1} denominator will cause the gradients to be very small

Vanishing gradients: illustration



We have:

$$\begin{array}{rcll} \mathcal{B}^L & = & \Phi'(\zeta^L)\cdot\mathcal{C}'(\alpha^L,\rho) & \leq \frac{\mathcal{C}'(\alpha^L,\rho)}{4} \\ \mathcal{B}^{L-1} & = & \Phi'(\zeta^{L-1})\cdot\omega^L\cdot\mathcal{B}^L & \leq \omega^L\cdot\frac{\mathcal{C}'(\alpha^L,\rho)}{4^2} \\ & \vdots & & \\ \mathcal{B}^i & = & \Phi'(\zeta^i)\cdot\omega^{i+1}\cdot\mathcal{B}^{i+1} & \leq \left(\prod_{j=L}^{i+1}\omega^j\right)\cdot\frac{\mathcal{C}'(\alpha^L,\rho)}{4^{L-i+1}} \end{array}$$

- Except in the case where the weights are large, the 4^{L-i+1} denominator will cause the gradients to be very small
- This is a general problem for very deep neural networks

tanh activation function

- $\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$

tanh activation function

- $\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$
- $\tanh'(x) = 1 - (\tanh(x))^2$

tanh activation function

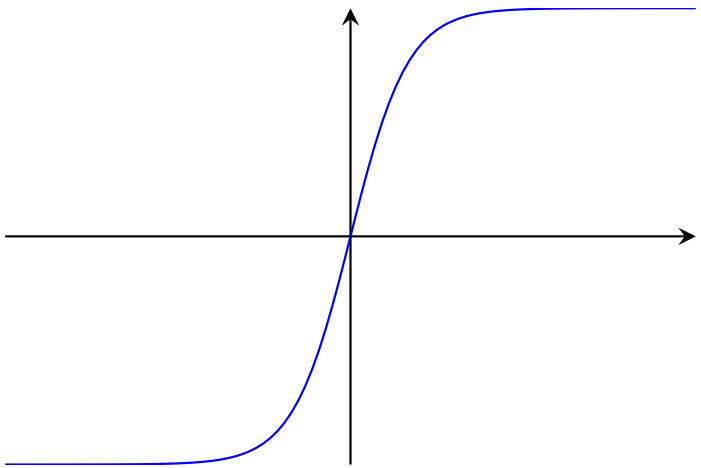
- $\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$
- $\tanh'(x) = 1 - (\tanh(x))^2$
- Centered around 0

tanh activation function

- $\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$
- $\tanh'(x) = 1 - (\tanh(x))^2$
- Centered around 0
- There is still the vanishing gradient problem

tanh activation function

- $\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$
- $\tanh'(x) = 1 - (\tanh(x))^2$
- Centered around 0
- There is still the vanishing gradient problem



Rectified Linear Unit (ReLU) activation function

- $\mathrm{ReLU}(x) = \max(0, x)$

Rectified Linear Unit (ReLU) activation function

- $\mathrm{ReLU}(x) = \max(0, x)$
- $\mathrm{ReLU}'(x) = \mathbb{1}_{\{x>0\}}$ **(debatable)**

Rectified Linear Unit (ReLU) activation function

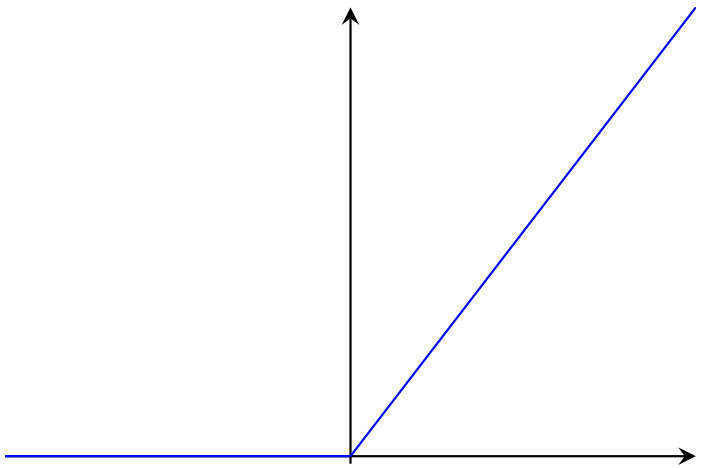
- $\mathrm{ReLU}(x) = \max(0, x)$
- $\mathrm{ReLU}'(x) = \mathbb{1}_{\{x>0\}}$ **(debatable)**
- A. Krizhevsky: *ImageNet Classification with Deep Convolutional Neural Networks* (2012): image classifier trained six times faster with ReLU than with tanh

Rectified Linear Unit (ReLU) activation function

- $\mathrm{ReLU}(x) = \max(0, x)$
- $\mathrm{ReLU}'(x) = \mathbb{1}_{\{x>0\}}$ **(debatable)**
- A. Krizhevsky: *ImageNet Classification with Deep Convolutional Neural Networks* (2012): image classifier trained six times faster with ReLU than with tanh
- Issue with **dead neurons**

Rectified Linear Unit (ReLU) activation function

- $\mathrm{ReLU}(x) = \max(0, x)$
- $\mathrm{ReLU}'(x) = \mathbb{1}_{\{x>0\}}$ **(debatable)**
- A. Krizhevsky: *ImageNet Classification with Deep Convolutional Neural Networks* (2012): image classifier trained six times faster with ReLU than with tanh
- Issue with **dead neurons**



On dead neurons

- Assume the weights and bias of a neuron are such that $\omega^{\mathrm{T}}\alpha + \beta < 0$ during the training phase

On dead neurons

- Assume the weights and bias of a neuron are such that $\omega^{\mathrm{T}}\alpha + \beta < 0$ during the training phase
- This can happen when:
  - The input to the neuron is always positive and the weights are negative

On dead neurons

- Assume the weights and bias of a neuron are such that $\omega^{\mathrm{T}}\alpha + \beta < 0$ during the training phase
- This can happen when:
  - The input to the neuron is always positive and the weights are negative
  - The weights are small and the bias is negative
    - Such a case could occur if a large learning rate is used

On dead neurons

- Assume the weights and bias of a neuron are such that $\omega^{\mathrm{T}}\alpha + \beta < 0$ during the training phase
- This can happen when:
  - The input to the neuron is always positive and the weights are negative
  - The weights are small and the bias is negative
    - Such a case could occur if a large learning rate is used
- The activation of this neuron will always be 0

On dead neurons

- Assume the weights and bias of a neuron are such that $\omega^{\mathrm{T}}\alpha + \beta < 0$ during the training phase
- This can happen when:
  - The input to the neuron is always positive and the weights are negative
  - The weights are small and the bias is negative
    - Such a case could occur if a large learning rate is used
- The activation of this neuron will always be 0
- The gradient for this neuron will always be 0

On dead neurons

- Assume the weights and bias of a neuron are such that $\omega^{\mathrm{T}}\alpha + \beta < 0$ during the training phase
- This can happen when:
  - The input to the neuron is always positive and the weights are negative
  - The weights are small and the bias is negative
    - Such a case could occur if a large learning rate is used
- The activation of this neuron will always be 0
- The gradient for this neuron will always be 0
- The neuron has no contribution and cannot be updated

Leaky ReLU

- $\text{LeakyReLU}(x) = \mathbb{1}_{\{x<0\}} \cdot \alpha x + \mathbb{1}_{\{x>0\}} \cdot x$

Leaky ReLU

- $\text{LeakyReLU}(x) = \mathbb{1}_{\{x<0\}} \cdot \alpha x + \mathbb{1}_{\{x>0\}} \cdot x$
- $\text{LeakyReLU}'(x) = \alpha \mathbb{1}_{\{x<0\}} + \mathbb{1}_{\{x>0\}}$

Leaky ReLU

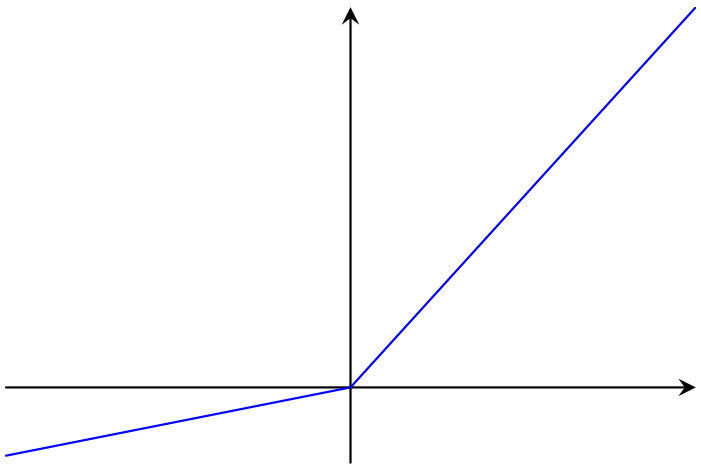
- $\text{LeakyReLU}(x) = \mathbb{1}_{\{x<0\}} \cdot \alpha x + \mathbb{1}_{\{x>0\}} \cdot x$
- $\text{LeakyReLU}'(x) = \alpha \mathbb{1}_{\{x<0\}} + \mathbb{1}_{\{x>0\}}$
- The value of α is small (generally 0.01)

Leaky ReLU

- $\mathrm{LeakyReLU}(x) = \mathbb{1}_{\{x<0\}} \cdot \alpha x + \mathbb{1}_{\{x>0\}} \cdot x$
- $\mathrm{LeakyReLU}'(x) = \alpha \mathbb{1}_{\{x<0\}} + \mathbb{1}_{\{x>0\}}$
- The value of α is small (generally 0.01)
- Solves the issue of dead neurons

Leaky ReLU

- $\mathrm{LeakyReLU}(x) = \mathbb{1}_{\{x<0\}} \cdot \alpha x + \mathbb{1}_{\{x>0\}} \cdot x$
- $\mathrm{LeakyReLU}'(x) = \alpha \mathbb{1}_{\{x<0\}} + \mathbb{1}_{\{x>0\}}$
- The value of α is small (generally 0.01)
- Solves the issue of dead neurons



Recommendations on activation functions

- Many activation functions
  - In particular, many variants of ReLU

Recommendations on activation functions

- Many activation functions
  - In particular, many variants of ReLU
- Choose one that is popular and stick to it
  - Uniformly on all layers

Recommendations on activation functions

- Many activation functions
  - In particular, many variants of ReLU
- Choose one that is popular and stick to it
  - Uniformly on all layers
  - **Except for output layer**: it is standard to use a linear activation for regression problems

Recommendations on activation functions

- Many activation functions
  - In particular, many variants of ReLU
- Choose one that is popular and stick to it
  - Uniformly on all layers
  - **Except for output layer**: it is standard to use a linear activation for regression problems
  - Some seem more convenient depending on the amount of data at hand and the time that can be spent training the network

Outline

Outline of this section

What should the goal of training a neural network be?

- At each epoch, weights are updated to reduce the **training error**

What should the goal of training a neural network be?

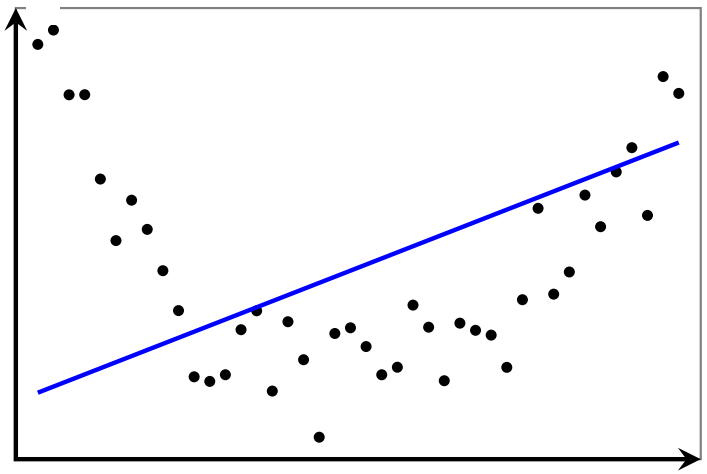
- At each epoch, weights are updated to reduce the **training error**
- The goal of training should be to obtain a network that performs well on **unknown** inputs
  - Given a fresh set of samples, the error on this set should be as low as possible

What should the goal of training a neural network be?

- At each epoch, weights are updated to reduce the **training error**
- The goal of training should be to obtain a network that performs well on **unknown** inputs
  - Given a fresh set of samples, the error on this set should be as low as possible
- The goal of training should be to minimize this **test error**

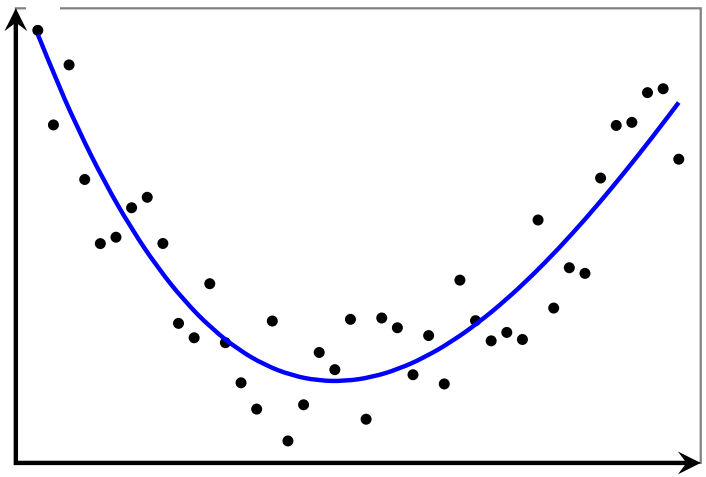
What can happen when training a neural network

- At the beginning, the neural network is completely off and the training error is large, as the network parameters have not been updated yet (**underfitting**)



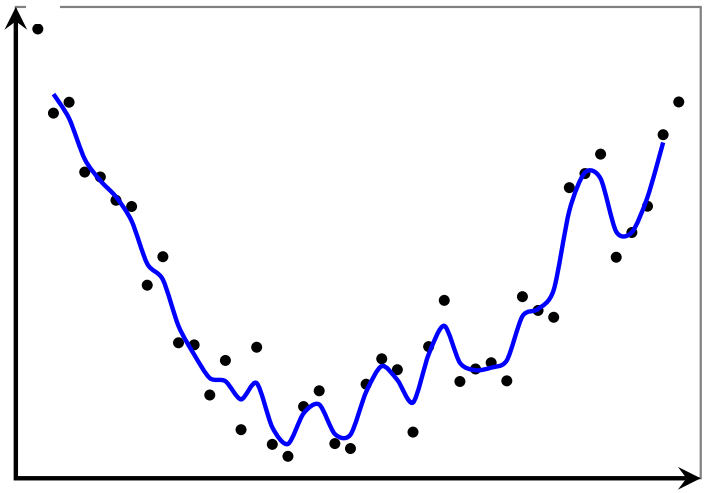
What can happen when training a neural network

- At the beginning, the neural network is completely off and the training error is large, as the network parameters have not been updated yet (**underfitting**)
- After some time, the training error has been reduced and the network starts behaving nicely



What can happen when training a neural network

- At the beginning, the neural network is completely off and the training error is large, as the network parameters have not been updated yet (**underfitting**)
- After some time, the training error has been reduced and the network starts behaving nicely
- As we keep training the network, the training error is close to 0, but the test error will increase: the network is **overfitting** the data



Detecting overfitting

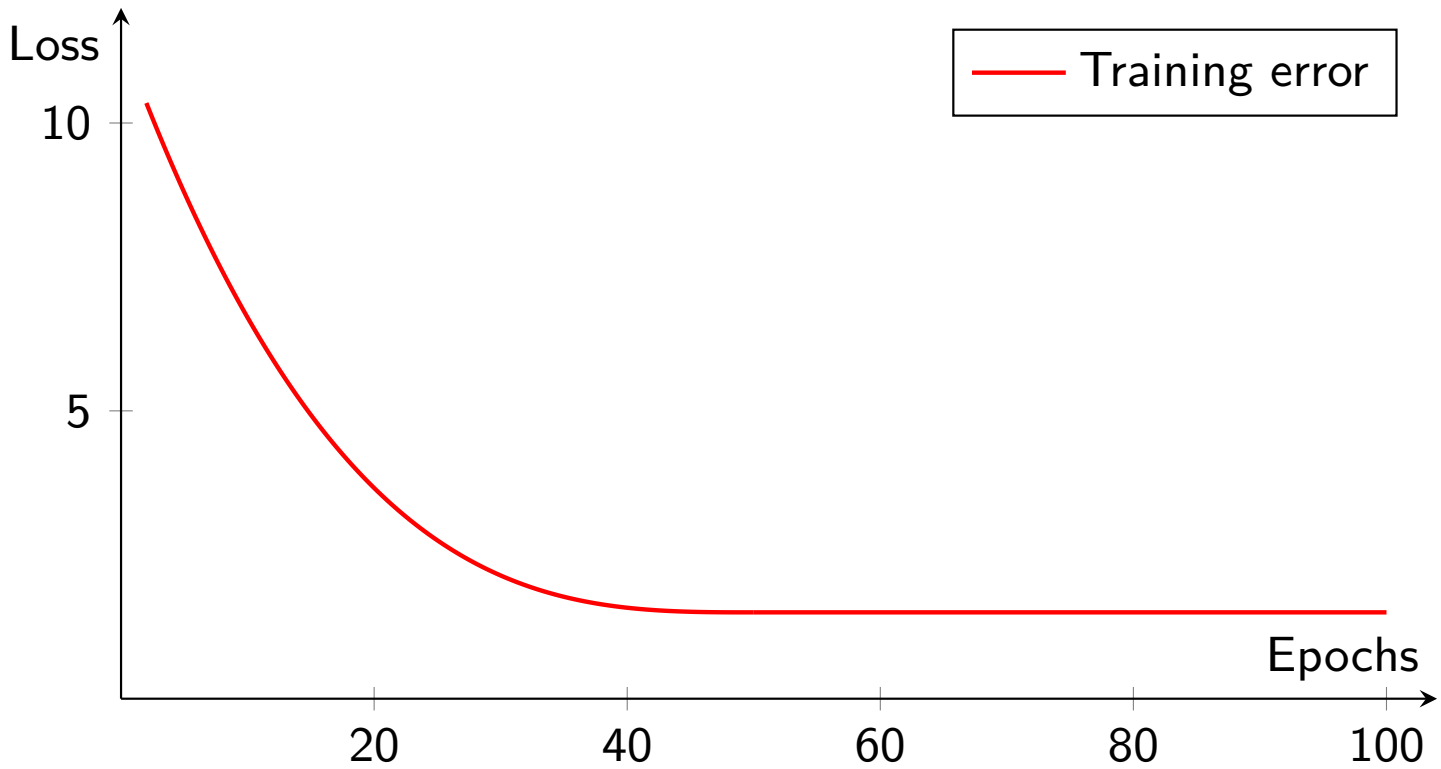
- Separate samples into (at least) two sets:
  - The **training set**, which is used to learn the parameters of the neural network
  - The **validation set**, which is used to evaluate the network

Detecting overfitting

- Separate samples into (at least) two sets:
  - The **training set**, which is used to learn the parameters of the neural network
  - The **validation set**, which is used to evaluate the network
- When the loss on the validation set stops decreasing, the network is probably overfitting the data ("learning noise")

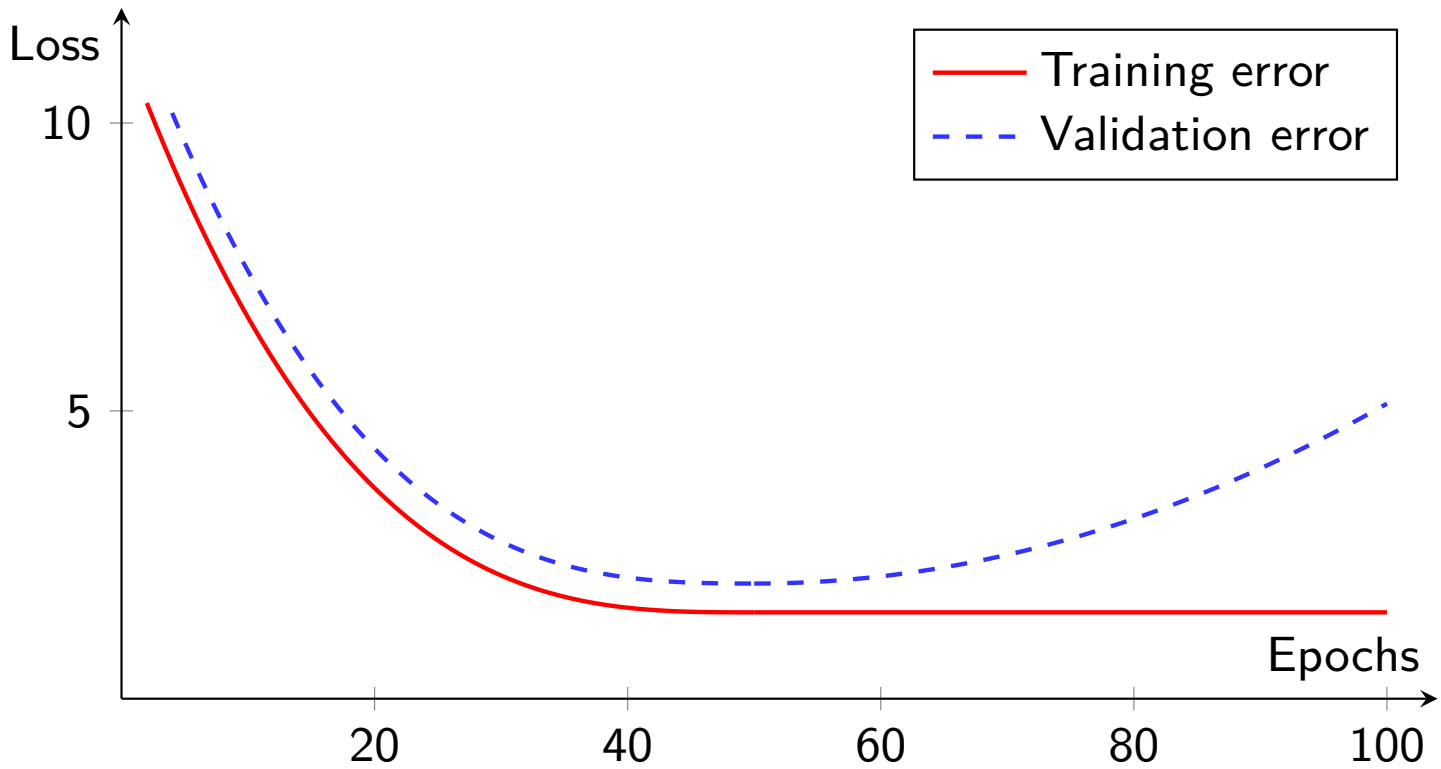
Detecting overfitting

- Separate samples into (at least) two sets:
  - The **training set**, which is used to learn the parameters of the neural network
  - The **validation set**, which is used to evaluate the network
- When the loss on the validation set stops decreasing, the network is probably overfitting the data ("learning noise")



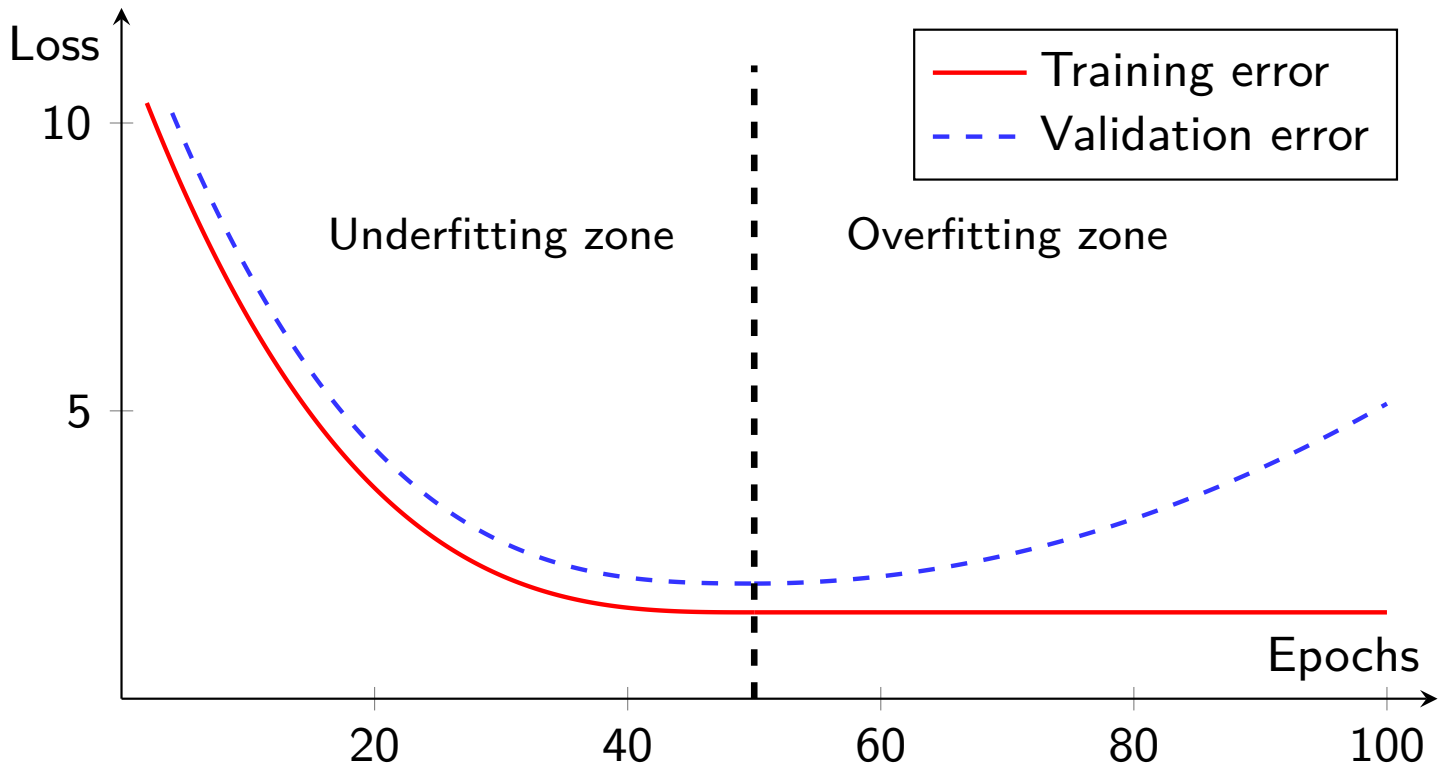
Detecting overfitting

- Separate samples into (at least) two sets:
  - The **training set**, which is used to learn the parameters of the neural network
  - The **validation set**, which is used to evaluate the network
- When the loss on the validation set stops decreasing, the network is probably overfitting the data ("learning noise")



Detecting overfitting

- Separate samples into (at least) two sets:
  - The **training set**, which is used to learn the parameters of the neural network
  - The **validation set**, which is used to evaluate the network
- When the loss on the validation set stops decreasing, the network is probably overfitting the data ("learning noise")



Regularization techniques

- Techniques designed to ensure the gap between training and generalization errors is not too large

Regularization techniques

- Techniques designed to ensure the gap between training and generalization errors is not too large
- General principle: impose preferences on the optimal weights that are computed

Regularization techniques

- Techniques designed to ensure the gap between training and generalization errors is not too large
- General principle: impose preferences on the optimal weights that are computed
  - By constraining the form of these weights (complexity and/or amplitude) (e.g. **weight penalties**)

Regularization techniques

- Techniques designed to ensure the gap between training and generalization errors is not too large
- General principle: impose preferences on the optimal weights that are computed
  - By constraining the form of these weights (complexity and/or amplitude) (e.g. **weight penalties**)
  - By restricting the computation resources (e.g. **early stopping**)

Regularization techniques

- Techniques designed to ensure the gap between training and generalization errors is not too large
- General principle: impose preferences on the optimal weights that are computed
  - By constraining the form of these weights (complexity and/or amplitude) (e.g. **weight penalties**)
  - By restricting the computation resources (e.g. **early stopping**)
  - By modifying the way weights are updated (e.g. **dropout**)

Regularization techniques

- Techniques designed to ensure the gap between training and generalization errors is not too large
- General principle: impose preferences on the optimal weights that are computed
  - By constraining the form of these weights (complexity and/or amplitude) (e.g. **weight penalties**)
  - By restricting the computation resources (e.g. **early stopping**)
  - By modifying the way weights are updated (e.g. **dropout**)
- A lot of ongoing research on this topic

Regularization techniques

- Techniques designed to ensure the gap between training and generalization errors is not too large
- General principle: impose preferences on the optimal weights that are computed
  - By constraining the form of these weights (complexity and/or amplitude) (e.g. **weight penalties**)
  - By restricting the computation resources (e.g. **early stopping**)
  - By modifying the way weights are updated (e.g. **dropout**)
- A lot of ongoing research on this topic

Importance of regularization

In practice, deep learning architectures that give good results are large, and have had proper regularization techniques applied to them

Outline of this section

Penalties on weights

- It is standard to impose penalties on weights, but not on biases
  - Large weights make minor changes in inputs have a major effect
  - Biases are not affected by these minor changes

Penalties on weights

- It is standard to impose penalties on weights, but not on biases
  - Large weights make minor changes in inputs have a major effect
  - Biases are not affected by these minor changes
- For the sake of clarity, we assume we have a network with no biases

Penalties on weights

- It is standard to impose penalties on weights, but not on biases
  - Large weights make minor changes in inputs have a major effect
  - Biases are not affected by these minor changes
- For the sake of clarity, we assume we have a network with no biases
- Principle: update the loss function to include a norm penalty

$$\mathcal{E}'(S, \theta) \stackrel{\text{def}}{=} \mathcal{E}(S, \theta) + \kappa \mathcal{P}(\theta)$$

Penalties on weights

- It is standard to impose penalties on weights, but not on biases
  - Large weights make minor changes in inputs have a major effect
  - Biases are not affected by these minor changes
- For the sake of clarity, we assume we have a network with no biases
- Principle: update the loss function to include a norm penalty

$$\mathcal{E}'(S, \theta) \stackrel{\text{def}}{=} \mathcal{E}(S, \theta) + \kappa \mathcal{P}(\theta)$$

  - κ is a hyperparameter used to specify how much importance the penalty term should have

Penalties on weights

- It is standard to impose penalties on weights, but not on biases
  - Large weights make minor changes in inputs have a major effect
  - Biases are not affected by these minor changes
- For the sake of clarity, we assume we have a network with no biases
- Principle: update the loss function to include a norm penalty

$$\mathcal{E}'(S, \theta) \stackrel{\text{def}}{=} \mathcal{E}(S, \theta) + \kappa \mathcal{P}(\theta)$$

  - κ is a hyperparameter used to specify how much importance the penalty term should have
  - If it is too low, the network will probably overfit the data

Penalties on weights

- It is standard to impose penalties on weights, but not on biases
  - Large weights make minor changes in inputs have a major effect
  - Biases are not affected by these minor changes
- For the sake of clarity, we assume we have a network with no biases
- Principle: update the loss function to include a norm penalty

$$\mathcal{E}'(S, \theta) \stackrel{\text{def}}{=} \mathcal{E}(S, \theta) + \kappa \mathcal{P}(\theta)$$

  - κ is a hyperparameter used to specify how much importance the penalty term should have
  - If it is too low, the network will probably overfit the data
  - If it is too large, the network will probably underfit the data

L^2 regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \frac{\|\theta\|_2^2}{2}$

L^2 regularization

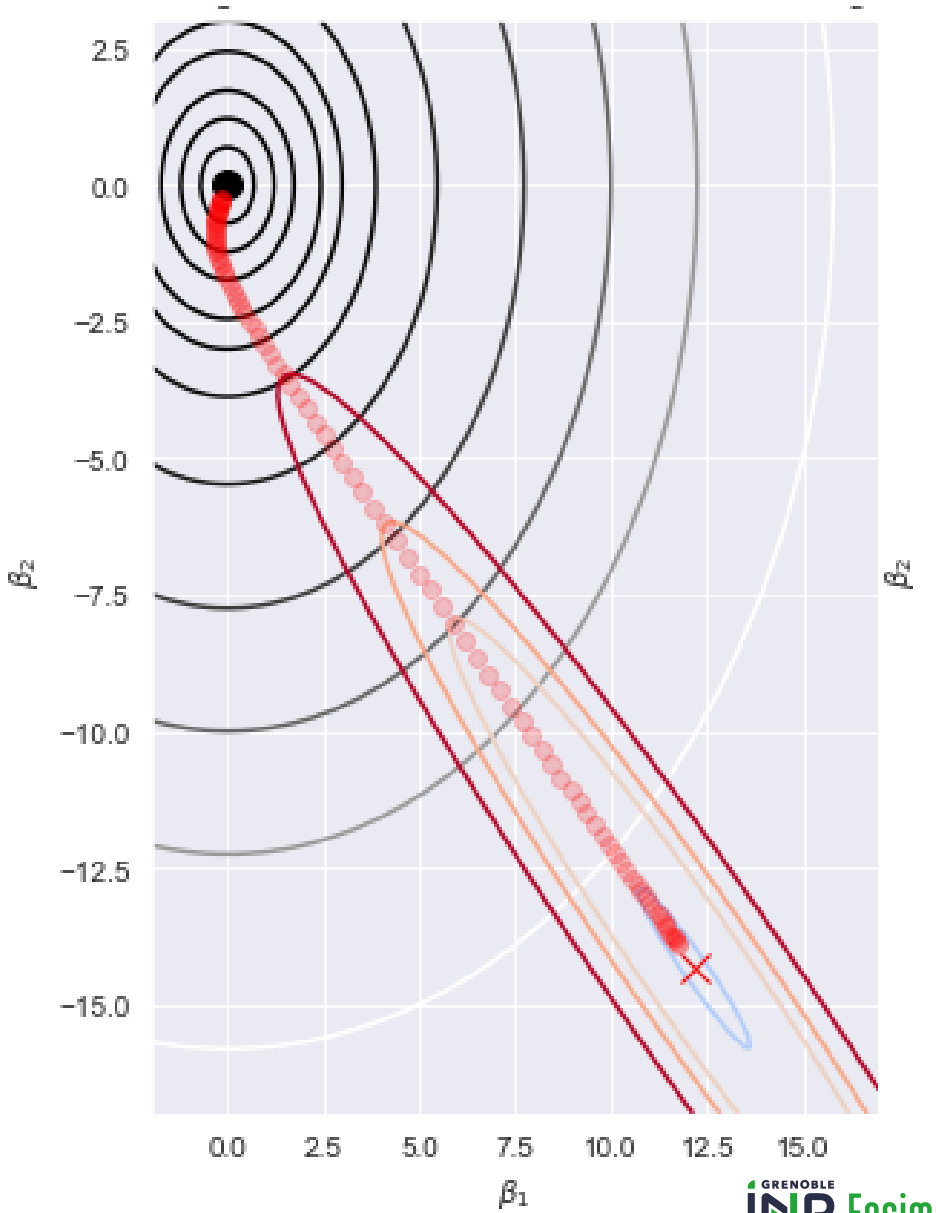
- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \frac{\|\theta\|_2^2}{2}$
- Also known as Tikhonov regularization or ridge regularization

L^2 regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \frac{\|\theta\|_2^2}{2}$
- Also known as Tikhonov regularization or ridge regularization
- Gradient descent rule with a fixed learning rate becomes $\theta \leftarrow (1 - \eta\kappa)\theta - \eta\nabla_\theta \mathcal{E}(S, \theta)$

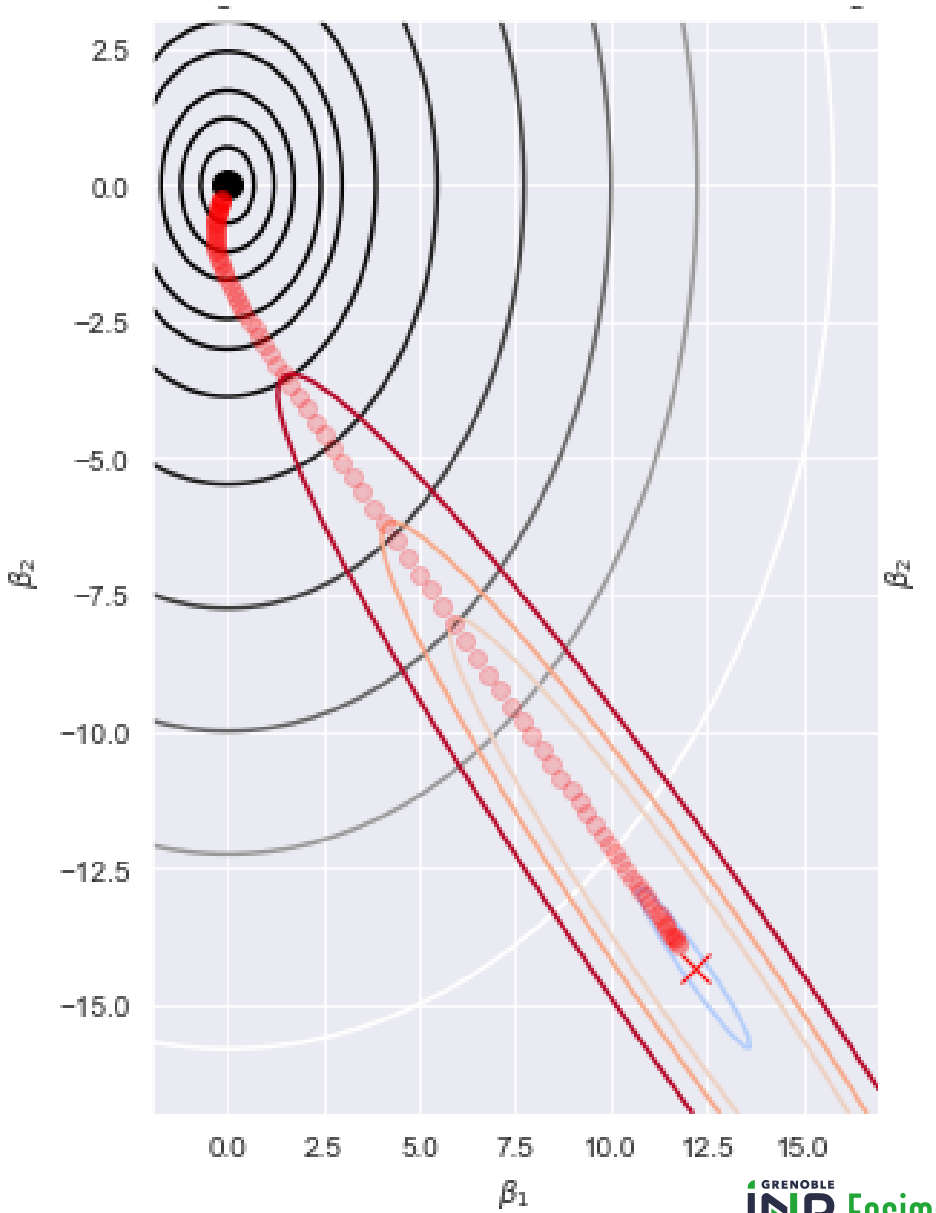
L^2 regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \frac{\|\theta\|_2^2}{2}$
- Also known as Tikhonov regularization or ridge regularization
- Gradient descent rule with a fixed learning rate becomes $\theta \leftarrow (1 - \eta\kappa)\theta - \eta\nabla_\theta \mathcal{E}(S, \theta)$
- Tends to output networks with small weights
  - Illustration: ©F. Bourgey



L^2 regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \frac{\|\theta\|_2^2}{2}$
- Also known as Tikhonov regularization or ridge regularization
- Gradient descent rule with a fixed learning rate becomes $\theta \leftarrow (1 - \eta\kappa)\theta - \eta\nabla_\theta \mathcal{E}(S, \theta)$
- Tends to output networks with small weights
  - Illustration: ©F. Bourgey
- Irrelevant inputs are still taken into account, with small weights



L^1 regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \|\theta\|_1$

L^1 regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \|\theta\|_1$
- Also called **Lasso regularization**

L^1 regularization

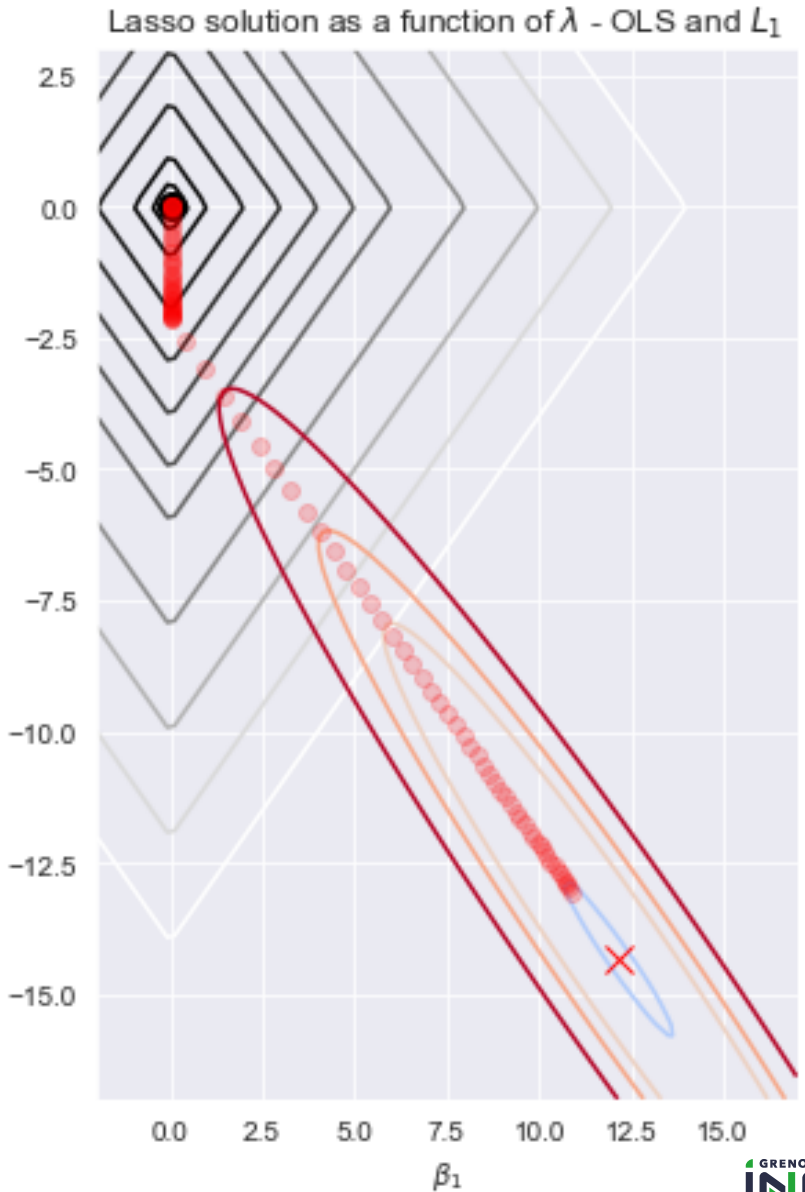
- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \|\theta\|_1$
- Also called **Lasso regularization**
- Gradient descent rule becomes

  $\theta \leftarrow \theta - \eta\kappa \cdot \text{sign}(\theta) - \eta\nabla_\theta \mathcal{E}(S, \theta)$

L^1 regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \|\theta\|_1$
- Also called **Lasso regularization**
- Gradient descent rule becomes

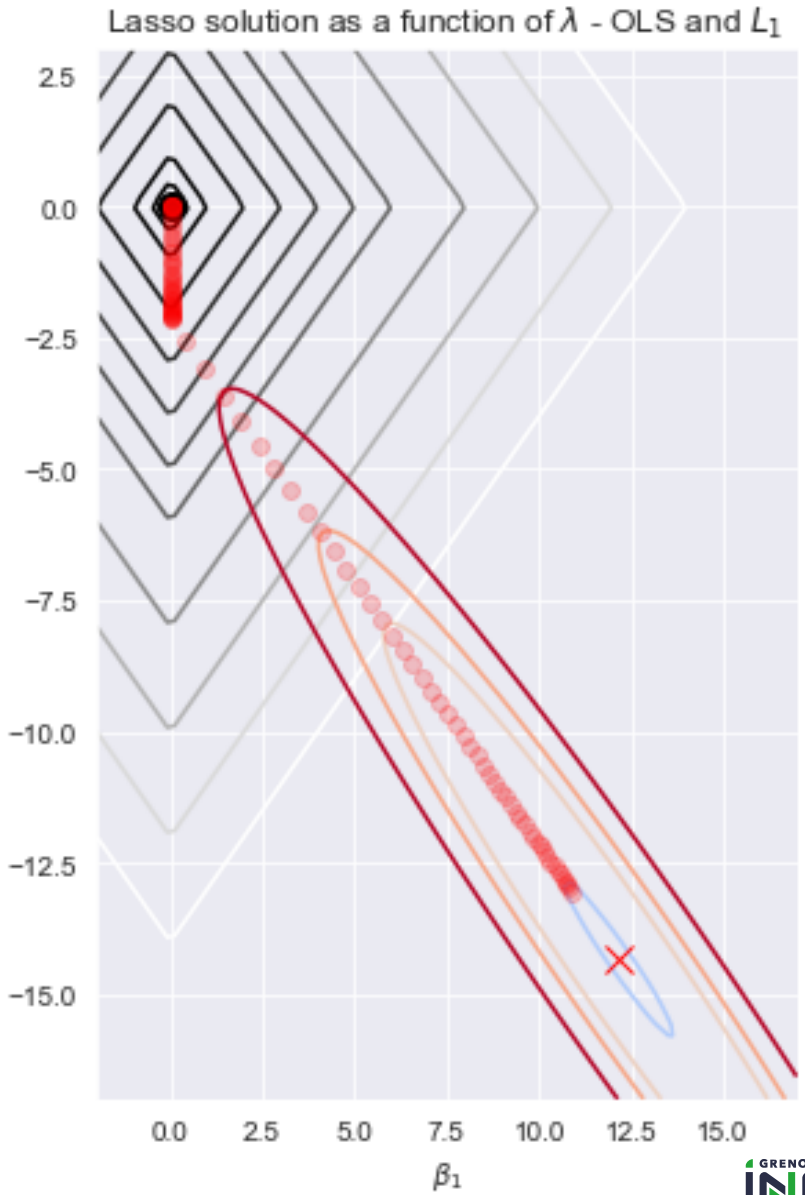
  $\theta \leftarrow \theta - \eta\kappa \cdot \text{sign}(\theta) - \eta\nabla_\theta \mathcal{E}(S, \theta)$
- Tends to output sparse networks (many weights equal to 0)
  - Illustration: ©F. Bourgey



L^1 regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \|\theta\|_1$
- Also called **Lasso regularization**
- Gradient descent rule becomes

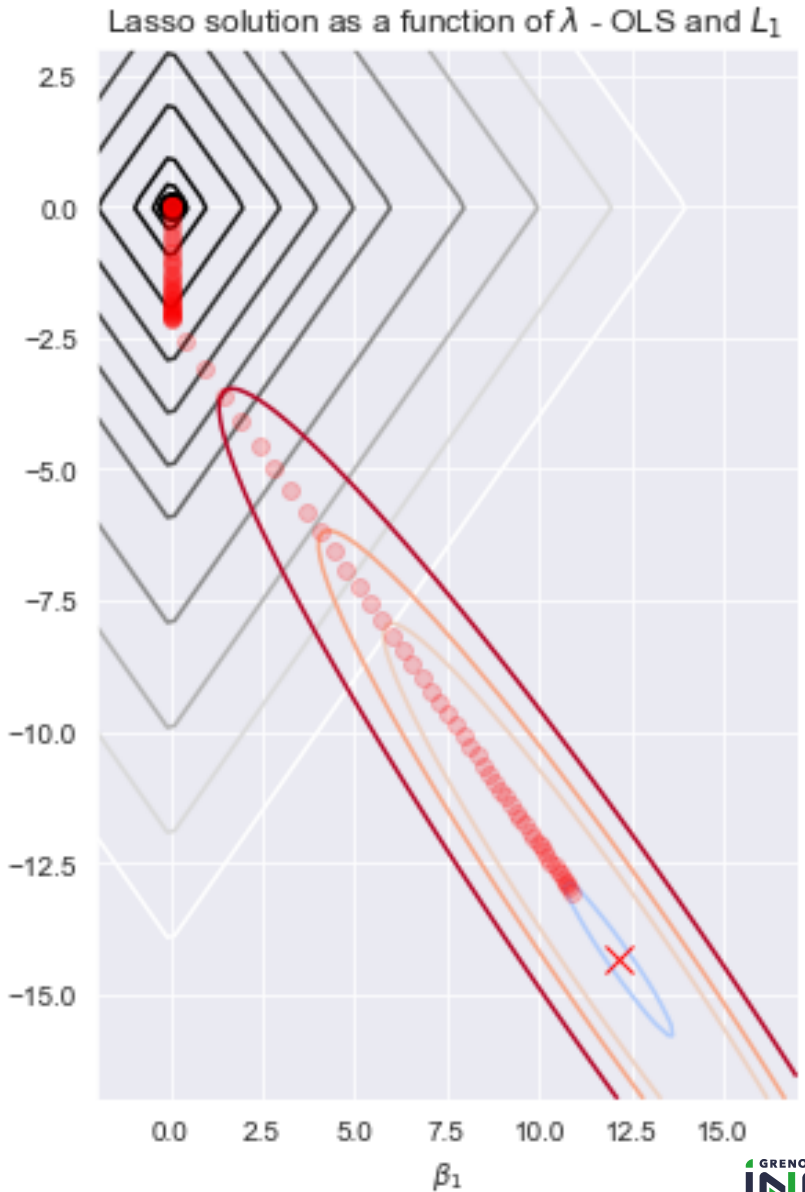
  $\theta \leftarrow \theta - \eta\kappa \cdot \text{sign}(\theta) - \eta\nabla_\theta \mathcal{E}(S, \theta)$
- Tends to output sparse networks (many weights equal to 0)
  - Illustration: ©F. Bourgey
- Can be used for **feature selection**



L^1 regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \|\theta\|_1$
- Also called **Lasso regularization**
- Gradient descent rule becomes

  $\theta \leftarrow \theta - \eta\kappa \cdot \text{sign}(\theta) - \eta\nabla_\theta \mathcal{E}(S, \theta)$
- Tends to output sparse networks (many weights equal to 0)
  - Illustration: ©F. Bourgey
- Can be used for **feature selection**
- Issue for problems with many dimensions but few samples



Elastic net regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \kappa' \|\theta\|_1 + \frac{(1-\kappa')}{2} \cdot \|\theta\|_2^2$, for $0 \leq \kappa' \leq 1$

Elastic net regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \kappa' \|\theta\|_1 + \frac{(1-\kappa')}{2} \cdot \|\theta\|_2^2$, for $0 \leq \kappa' \leq 1$
- Convex combination of L^1 and L^2 regularization

Elastic net regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \kappa' \|\theta\|_1 + \frac{(1-\kappa')}{2} \cdot \|\theta\|_2^2$, for $0 \leq \kappa' \leq 1$
- Convex combination of L^1 and L^2 regularization
- Introduced in 2005 to overcome limitations of L^1 regularization

Elastic net regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \kappa' \|\theta\|_1 + \frac{(1-\kappa')}{2} \cdot \|\theta\|_2^2$, for $0 \leq \kappa' \leq 1$
- Convex combination of L^1 and L^2 regularization
- Introduced in 2005 to overcome limitations of L^1 regularization
- Often a good default choice

Elastic net regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \kappa' \|\theta\|_1 + \frac{(1-\kappa')}{2} \cdot \|\theta\|_2^2$, for $0 \leq \kappa' \leq 1$
- Convex combination of L^1 and L^2 regularization
- Introduced in 2005 to overcome limitations of L^1 regularization
- Often a good default choice
  - But there is a new hyperparameter to tune

A layer for L^2 regularization

- Constructor parameters:
  - Hyperparameter κ
  - Underlying layer

A layer for L^2 regularization

- Constructor parameters:
  - Hyperparameter κ
  - Underlying layer
- Forward propagation
  - Invoke forward propagation on the underlying layer

A layer for L^2 regularization

- Constructor parameters:
  - Hyperparameter κ
  - Underlying layer
- Forward propagation
  - Invoke forward propagation on the underlying layer
- Backpropagation
  - Invoke backpropagation and compute weight gradients on the underlying layer
  - Multiply weights of the underlying layer by the penalty coefficient κ
  - Add the result to the weight gradient of the underlying layer

A layer for L^2 regularization

- Constructor parameters:
  - Hyperparameter κ
  - Underlying layer
- Forward propagation
  - Invoke forward propagation on the underlying layer
- Backpropagation
  - Invoke backpropagation and compute weight gradients on the underlying layer
  - Multiply weights of the underlying layer by the penalty coefficient κ
  - Add the result to the weight gradient of the underlying layer
- Parameter update
  - Invoke parameter update on the underlying layer (with the updated weight gradients)

On weight decay

- Another regularization technique designed to compute networks with small weights

On weight decay

- Another regularization technique designed to compute networks with small weights
- Update rule: $\theta \leftarrow (1 - \lambda)\theta - \eta \nabla_\theta \mathcal{E}(S, \theta)$

On weight decay

- Another regularization technique designed to compute networks with small weights
- Update rule: $\theta \leftarrow (1 - \lambda)\theta - \eta \nabla_\theta \mathcal{E}(S, \theta)$
- Isn't this the same as L^2-regularization?

On weight decay

- Another regularization technique designed to compute networks with small weights
- Update rule: $\theta \leftarrow (1 - \lambda)\theta - \eta \nabla_\theta \mathcal{E}(S, \theta)$
- Isn't this the same as L^2-regularization?
  - For vanilla gradient descent, yes

On weight decay

- Another regularization technique designed to compute networks with small weights
- Update rule: $\theta \leftarrow (1 - \lambda)\theta - \eta \nabla_\theta \mathcal{E}(S, \theta)$
- Isn't this the same as L^2-regularization?
  - For vanilla gradient descent, yes
  - For adaptive gradient algorithms $\theta \leftarrow \theta + f(\theta, \nabla_\theta \mathcal{E}(S, \theta))$, no:

On weight decay

- Another regularization technique designed to compute networks with small weights
- Update rule: $\theta \leftarrow (1-\lambda)\theta - \eta\nabla_\theta \mathcal{E}(S,\theta)$
- Isn't this the same as L^2-regularization?
  - For vanilla gradient descent, yes
  - For adaptive gradient algorithms $\theta \leftarrow \theta + f(\theta, \nabla_\theta \mathcal{E}(S,\theta))$, no:
    - L^2-regularization produces $\theta \leftarrow \theta + f(\theta, \nabla_\theta \mathcal{E}(S,\theta) + \kappa\theta)$

On weight decay

- Another regularization technique designed to compute networks with small weights
- Update rule: $\theta \leftarrow (1-\lambda)\theta - \eta\nabla_\theta \mathcal{E}(S,\theta)$
- Isn't this the same as L^2-regularization?
  - For vanilla gradient descent, yes
  - For adaptive gradient algorithms $\theta \leftarrow \theta + f(\theta, \nabla_\theta \mathcal{E}(S,\theta))$, no:
    - L^2-regularization produces $\theta \leftarrow \theta + f(\theta, \nabla_\theta \mathcal{E}(S,\theta) + \kappa\theta)$
    - Weight decay produces $\theta \leftarrow (1-\lambda)\theta + f(\theta, \nabla_\theta \mathcal{E}(S,\theta))$

On weight decay

- Another regularization technique designed to compute networks with small weights
- Update rule: $\theta \leftarrow (1-\lambda)\theta - \eta\nabla_\theta \mathcal{E}(S,\theta)$
- Isn't this the same as L^2-regularization?
  - For vanilla gradient descent, yes
  - For adaptive gradient algorithms $\theta \leftarrow \theta + f(\theta, \nabla_\theta \mathcal{E}(S,\theta))$, no:
    - L^2-regularization produces $\theta \leftarrow \theta + f(\theta, \nabla_\theta \mathcal{E}(S,\theta) + \kappa\theta)$
    - Weight decay produces $\theta \leftarrow (1-\lambda)\theta + f(\theta, \nabla_\theta \mathcal{E}(S,\theta))$
- Adam appears to perform much better with weight decay than with L^2-regularization (Loshchilov & Hutter - Decoupled weight decay regularization. 2019)

Outline of this section

Early stopping

- Quite popular: unobtrusive, simple to implement, can be used with other techniques

Early stopping

- Quite popular: unobtrusive, simple to implement, can be used with other techniques
- Principle: train while monitoring validation error; stop when validation error has not improved for some time

Early stopping

- Quite popular: unobtrusive, simple to implement, can be used with other techniques
- Principle: train while monitoring validation error; stop when validation error has not improved for some time

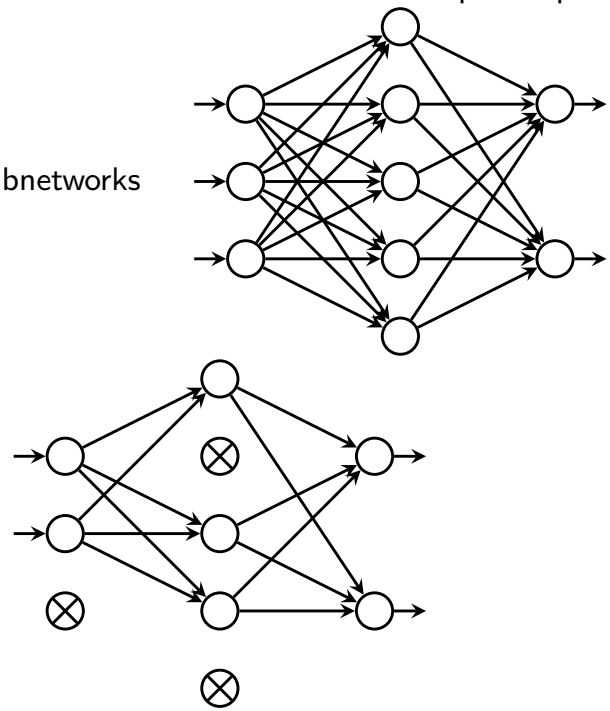
Input: n, number of steps between validation error evaluations
Input: p, number of observations of worsening validation errors before stop
Input: θ_0, initial parameters

$\theta \leftarrow \theta_0$, $\theta^* \leftarrow \theta$;
bestError $\leftarrow +\infty$;
$j \leftarrow 0$;
while $j < p$ **do**
    Perform n updates of θ;
    $j \leftarrow j + 1$;
    if *ValidationError*(θ) $<$ *bestError* **then**
        bestError $\leftarrow$ ValidationError(θ);
        $j \leftarrow 0$;
        $\theta^* \leftarrow \theta$;
    end
end
return θ^*

Outline of this section

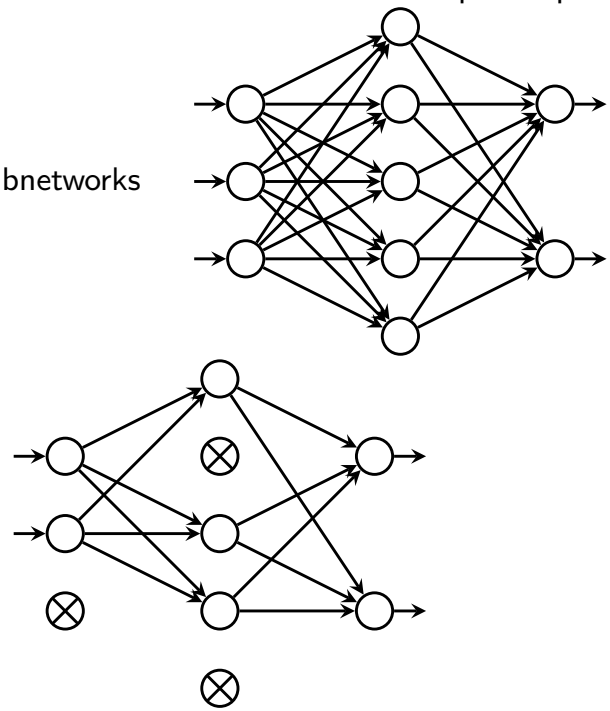
Dropout: intuition

- A network can be viewed as a compact representation of the set of all its subnetworks



Dropout: intuition

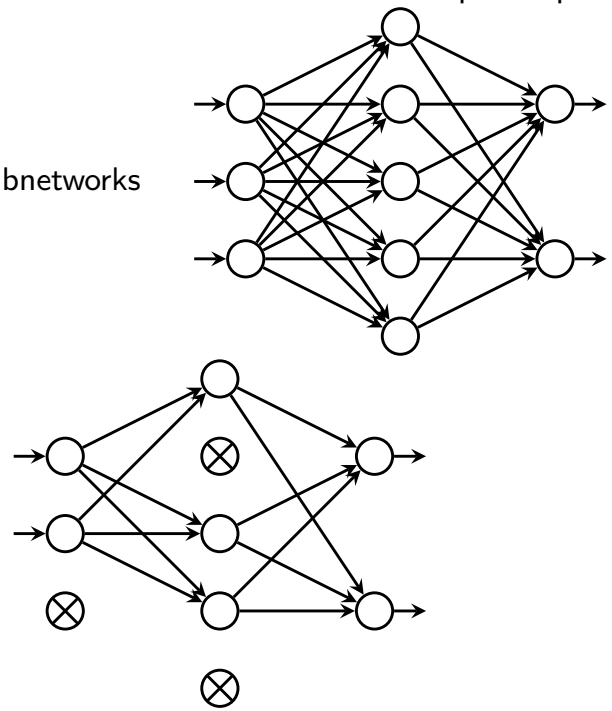
- A network can be viewed as a compact representation of the set of all its subnetworks



- These subnetworks are not independent: they share weights

Dropout: intuition

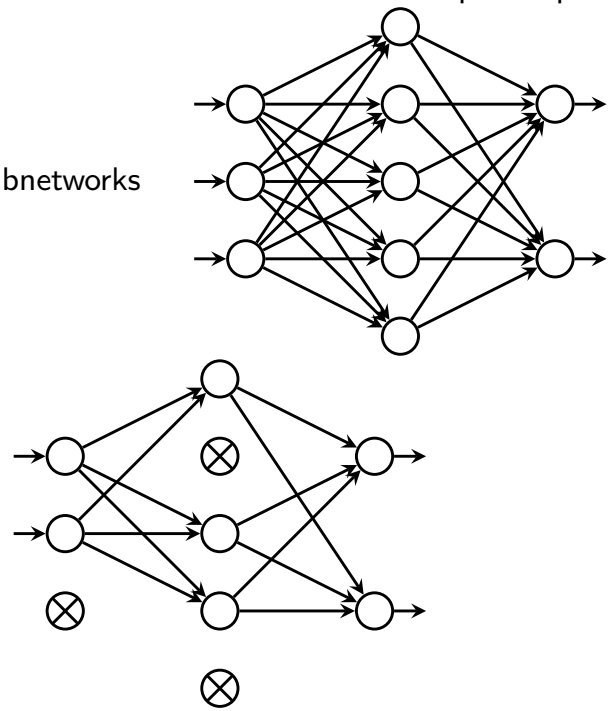
- A network can be viewed as a compact representation of the set of all its subnetworks



- These subnetworks are not independent: they share weights
- Why not train these subnetworks separately and average their predictions during the test phase?

Dropout: intuition

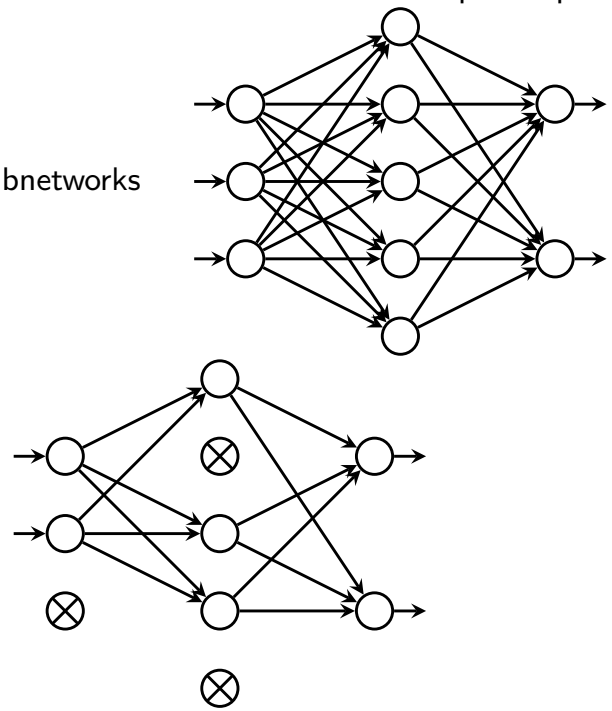
- A network can be viewed as a compact representation of the set of all its subnetworks



- These subnetworks are not independent: they share weights
- Why not train these subnetworks separately and average their predictions during the test phase?
- Issues:

Dropout: intuition

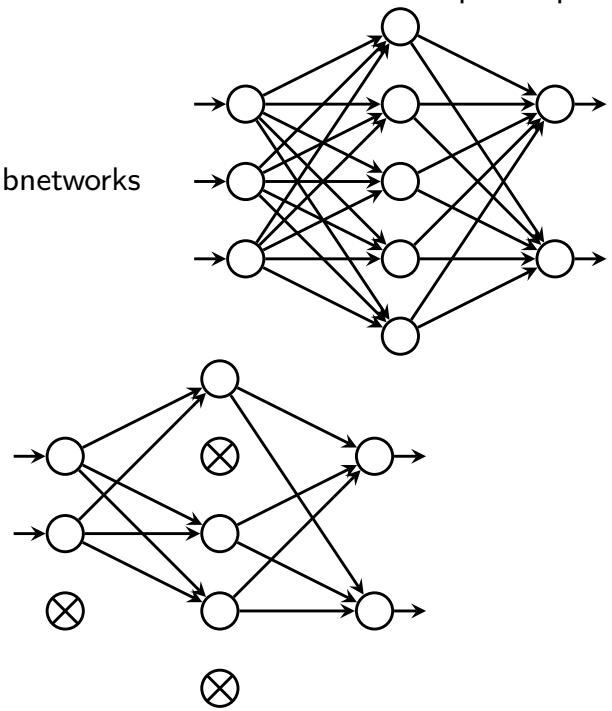
- A network can be viewed as a compact representation of the set of all its subnetworks



- These subnetworks are not independent: they share weights
- Why not train these subnetworks separately and average their predictions during the test phase?
- Issues:
  - Which subnetworks are trained and how?

Dropout: intuition

- A network can be viewed as a compact representation of the set of all its subnetworks



- These subnetworks are not independent: they share weights
- Why not train these subnetworks separately and average their predictions during the test phase?
- Issues:
  - Which subnetworks are trained and how?
  - How are their predictions averaged?

Practical dropout

- Training

Practical dropout

- Training
  - For each mini-batch, kill off some neurons, **except for output neurons**

Practical dropout

- Training
  - For each mini-batch, kill off some neurons, **except for output neurons**
  - Keep a neuron with probability p

Practical dropout

- Training
  - For each mini-batch, kill off some neurons, **except for output neurons**
  - Keep a neuron with probability p
  - In total, we will be training as many subnetworks as we consider mini-batches, each with a single update step

Practical dropout

- Training
  - For each mini-batch, kill off some neurons, **except for output neurons**
  - Keep a neuron with probability p
  - In total, we will be training as many subnetworks as we consider mini-batches, each with a single update step
- Testing

Practical dropout

- Training
  - For each mini-batch, kill off some neurons, **except for output neurons**
  - Keep a neuron with probability p
  - In total, we will be training as many subnetworks as we consider mini-batches, each with a single update step
- Testing
  - Averaging predictions over all subnetworks is not practical at all

Practical dropout

- Training
  - For each mini-batch, kill off some neurons, **except for output neurons**
  - Keep a neuron with probability p
  - In total, we will be training as many subnetworks as we consider mini-batches, each with a single update step
- Testing
  - Averaging predictions over all subnetworks is not practical at all
  - Approximation of the average: **weight scaling**

Practical dropout

- Training
  - For each mini-batch, kill off some neurons, **except for output neurons**
  - Keep a neuron with probability p
  - In total, we will be training as many subnetworks as we consider mini-batches, each with a single update step
- Testing
  - Averaging predictions over all subnetworks is not practical at all
  - Approximation of the average: **weight scaling**
  - Activation of layer i is $p \cdot \alpha^i$

Practical dropout

- Training
  - For each mini-batch, kill off some neurons, **except for output neurons**
  - Keep a neuron with probability p
  - In total, we will be training as many subnetworks as we consider mini-batches, each with a single update step
- Testing
  - Averaging predictions over all subnetworks is not practical at all
  - Approximation of the average: **weight scaling**
  - Activation of layer i is $p \cdot \alpha^i$
  - No theoretical argument for accuracy in the general case, but good results in practice

Inverted dropout

- Goal: Avoid scaling the activations of layers during test phase

Inverted dropout

- Goal: Avoid scaling the activations of layers during test phase
- How:
  - During training phase, scale activation by a factor $1/p$
  - During test phase, return normal activation on network
- Implementation
  - Create a specialized Dropout layer, to be inserted between standard layers

Inverted dropout

- Goal: Avoid scaling the activations of layers during test phase
- How:
  - During training phase, scale activation by a factor $1/p$
  - During test phase, return normal activation on network
- Implementation
  - Create a specialized Dropout layer, to be inserted between standard layers
  - Weights of the layer: $\Omega = \mathbf{Id}$

Inverted dropout

- Goal: Avoid scaling the activations of layers during test phase
- How:
  - During training phase, scale activation by a factor $1/p$
  - During test phase, return normal activation on network
- Implementation
  - Create a specialized Dropout layer, to be inserted between standard layers
  - Weights of the layer: $\Omega = \mathbf{Id}$
  - Bias of the layer: $\beta = \mathbf{0}$

Inverted dropout

- Goal: Avoid scaling the activations of layers during test phase
- How:
  - During training phase, scale activation by a factor $1/p$
  - During test phase, return normal activation on network
- Implementation
  - Create a specialized Dropout layer, to be inserted between standard layers
  - Weights of the layer: $\Omega = \mathbf{Id}$
  - Bias of the layer: $\beta = \mathbf{0}$
  - Activation **per mini-batch** and **per neuron**:

$$\Phi_D(x) = \begin{cases} \frac{x}{p} & \text{with probability } p \\ 0 & \text{with probability } 1-p \end{cases}$$

On forward and backpropagation with dropout

- Inverted dropout layer between layers j and $j+1$ acts as a mask $\mu \in \{0, 1/p\}^{n_j}$

On forward and backpropagation with dropout

- Inverted dropout layer between layers j and $j+1$ acts as a mask $\mu \in \{0, 1/p\}^{n_j}$
- Activations:

$$\alpha^D = \Phi_D(\left[\Omega^D\right]^{\mathrm{T}} \alpha^j)$$

On forward and backpropagation with dropout

- Inverted dropout layer between layers j and $j+1$ acts as a mask $\mu \in \{0, 1/p\}^{n_j}$
- Activations:

$$\begin{aligned} \alpha^D &= \Phi_D(\left[\Omega^D\right]^{\mathrm{T}} \alpha^j) \\ &= \alpha^j \odot \mu \end{aligned}$$

On forward and backpropagation with dropout

- Inverted dropout layer between layers j and $j+1$ acts as a mask $\mu \in \{0, 1/p\}^{n_j}$
- Activations:

$$\begin{aligned} \alpha^D &= \Phi_D(\left[\Omega^D\right]^{\mathrm{T}} \alpha^j) \\ &= \alpha^j \odot \mu \\ \alpha^{j+1} &= \Phi(\left[\Omega^{j+1}\right]^{\mathrm{T}} \alpha^D + \beta^{j+1}) \end{aligned}$$

On forward and backpropagation with dropout

- Inverted dropout layer between layers j and $j+1$ acts as a mask $\mu \in \{0, 1/p\}^{n_j}$
- Activations:

$$\begin{aligned}
\alpha^D &= \Phi_D(\left[\Omega^D\right]^{\mathrm{T}} \alpha^j) \\
&= \alpha^j \odot \mu \\
\alpha^{j+1} &= \Phi(\left[\Omega^{j+1}\right]^{\mathrm{T}} \alpha^D + \beta^{j+1}) \\
&= \Phi\left(\left[\Omega^{j+1}\right]^{\mathrm{T}} (\alpha^j \odot \mu) + \beta^{j+1}\right)
\end{aligned}$$

On forward and backpropagation with dropout

- Inverted dropout layer between layers j and $j+1$ acts as a mask $\mu \in \{0, 1/p\}^{n_j}$
- Activations:

$$\begin{aligned} \alpha^D &= \Phi_D(\left[\Omega^D\right]^{\mathrm{T}} \alpha^j) \\ &= \alpha^j \odot \mu \\ \alpha^{j+1} &= \Phi(\left[\Omega^{j+1}\right]^{\mathrm{T}} \alpha^D + \beta^{j+1}) \\ &= \Phi\left(\left[\Omega^{j+1}\right]^{\mathrm{T}} (\alpha^j \odot \mu) + \beta^{j+1}\right) \end{aligned}$$

- Backpropagation:

$$\mathcal{B}^D = \Phi'_D(\zeta^D) \odot \left[\mathcal{P}^{j+1}\right]^{\mathrm{T}}$$

On forward and backpropagation with dropout

- Inverted dropout layer between layers j and $j+1$ acts as a mask $\mu \in \{0, 1/p\}^{n_j}$
- Activations:

$$\begin{aligned} \alpha^D &= \Phi_D(\left[\Omega^D\right]^{\mathrm{T}} \alpha^j) \\ &= \alpha^j \odot \mu \\ \alpha^{j+1} &= \Phi(\left[\Omega^{j+1}\right]^{\mathrm{T}} \alpha^D + \beta^{j+1}) \\ &= \Phi\left(\left[\Omega^{j+1}\right]^{\mathrm{T}} (\alpha^j \odot \mu) + \beta^{j+1}\right) \end{aligned}$$

- Backpropagation:

$$\begin{aligned} \mathcal{B}^D &= \Phi'_D(\zeta^D) \odot \left[\mathcal{P}^{j+1}\right]^{\mathrm{T}} \\ &= \mu \odot \left[\mathcal{P}^{j+1}\right]^{\mathrm{T}} \end{aligned}$$

On forward and backpropagation with dropout

- Inverted dropout layer between layers j and $j+1$ acts as a mask $\mu \in \{0, 1/p\}^{n_j}$
- Activations:

$$\begin{aligned} \alpha^D &= \Phi_D(\left[\Omega^D\right]^{\mathrm{T}} \alpha^j) \\ &= \alpha^j \odot \mu \\ \alpha^{j+1} &= \Phi(\left[\Omega^{j+1}\right]^{\mathrm{T}} \alpha^D + \beta^{j+1}) \\ &= \Phi\left(\left[\Omega^{j+1}\right]^{\mathrm{T}} (\alpha^j \odot \mu) + \beta^{j+1}\right) \end{aligned}$$

- Backpropagation:

$$\begin{aligned} \mathcal{B}^D &= \Phi'_D(\zeta^D) \odot \left[\mathcal{P}^{j+1}\right]^{\mathrm{T}} \\ &= \mu \odot \left[\mathcal{P}^{j+1}\right]^{\mathrm{T}} \\ \mathcal{B}^j &= \Phi'(\zeta^j) \odot \left[\Omega^D \cdot \mathcal{B}^D\right] \end{aligned}$$

On forward and backpropagation with dropout

- Inverted dropout layer between layers j and $j+1$ acts as a mask $\mu \in \{0, 1/p\}^{n_j}$
- Activations:

$$\begin{aligned} \alpha^D &= \Phi_D(\left[\Omega^D\right]^{\mathrm{T}} \alpha^j) \\ &= \alpha^j \odot \mu \\ \alpha^{j+1} &= \Phi(\left[\Omega^{j+1}\right]^{\mathrm{T}} \alpha^D + \beta^{j+1}) \\ &= \Phi\left(\left[\Omega^{j+1}\right]^{\mathrm{T}} (\alpha^j \odot \mu) + \beta^{j+1}\right) \end{aligned}$$

- Backpropagation:

$$\begin{aligned} \mathcal{B}^D &= \Phi'_D(\zeta^D) \odot \left[\mathcal{P}^{j+1}\right]^{\mathrm{T}} \\ &= \mu \odot \left[\mathcal{P}^{j+1}\right]^{\mathrm{T}} \\ \mathcal{B}^j &= \Phi'(\zeta^j) \odot \left[\Omega^D \cdot \mathcal{B}^D\right] \\ &= \Phi'(\zeta^j) \odot \left(\mu \odot \left[\mathcal{P}^{j+1}\right]^{\mathrm{T}}\right) \end{aligned}$$

On forward and backpropagation with dropout

- Inverted dropout layer between layers j and $j+1$ acts as a mask $\mu \in \{0, 1/p\}^{n_j}$
- Activations:

$$\begin{aligned}
\alpha^D &= \Phi_D(\left[\Omega^D\right]^{\mathrm{T}} \alpha^j) \\
&= \alpha^j \odot \mu \\
\alpha^{j+1} &= \Phi(\left[\Omega^{j+1}\right]^{\mathrm{T}} \alpha^D + \beta^{j+1}) \\
&= \Phi\left(\left[\Omega^{j+1}\right]^{\mathrm{T}} (\alpha^j \odot \mu) + \beta^{j+1}\right)
\end{aligned}$$

- Backpropagation:

$$\begin{aligned}
\mathcal{B}^D &= \Phi'_D(\zeta^D) \odot \left[\mathcal{P}^{j+1}\right]^{\mathrm{T}} \\
&= \mu \odot \left[\mathcal{P}^{j+1}\right]^{\mathrm{T}} \\
\mathcal{B}^j &= \Phi'(\zeta^j) \odot \left[\Omega^D \cdot \mathcal{B}^D\right] \\
&= \Phi'(\zeta^j) \odot \left(\mu \odot \left[\mathcal{P}^{j+1}\right]^{\mathrm{T}}\right) \\
&= \mu \odot \left(\Phi'(\zeta^j) \odot \left[\mathcal{P}^{j+1}\right]^{\mathrm{T}}\right)
\end{aligned}$$