Neural networks: an introduction

UE de Master 2, AOS2
Fall 2023

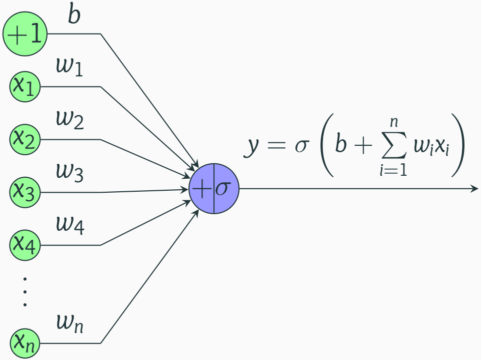
S. Rousseau

Feed forward neural network

What is a neuron?

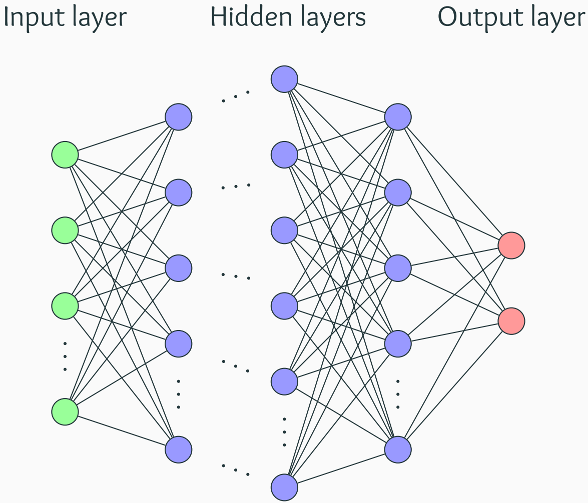
Neuron $=$ linear transform of the input followed by a non-linearity

- Input: $\mathbf{x} = (x_i)_{i=1\ldots n} \in \mathbb{R}^n$
- Scalar output: $y \in \mathbb{R}$
- $+1$ denotes the intercept (or bias) b
- The w_i's are called **weights**
- σ is nonlinear function called an **activation function**



Feed forward neural network

- Stacked collection of neurons arranged in layers
- Input layer nodes are not neurons
- Ouput layer neurons do not have necessarily an activation function



Input layer and first hidden layer

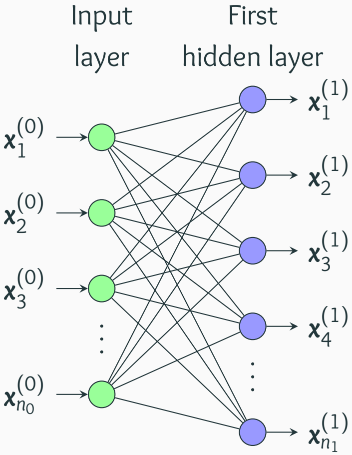
- Input: $\mathbf{x}^{(0)} = \left(\mathbf{x}_i^{(0)}\right)_{i=1\dots n_0} \in \mathbb{R}^{n_0}$
- Output: $\mathbf{x}^{(1)} = \left(\mathbf{x}_i^{(1)}\right)_{i=1\dots n_1} \in \mathbb{R}^{n_1}$

We have

$$\mathbf{x}_i^{(1)} = \sigma\left(\sum_{j=1}^{n_0} \boldsymbol{w}_{ij}^{(1)} \mathbf{x}_j^{(0)} + \boldsymbol{b}_i^{(1)}\right)$$

where

- $\boldsymbol{b}_i^{(1)}$ is the bias of neuron i (not displayed)
- $\boldsymbol{w}_{ij}^{(1)}$ is the j-th coefficient of i-th neuron
- Vector version $\mathbf{x}_i^{(1)} = \sigma\left(\left\langle \boldsymbol{w}_i^{(1)}, \mathbf{x}^{(0)} \right\rangle + \boldsymbol{b}_i^{(1)}\right)$

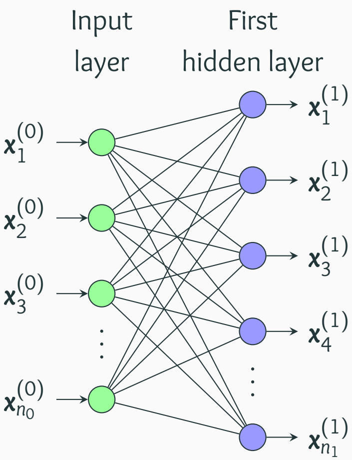


Input layer and first hidden layer

- Matrix version:

$$\mathbf{x}^{(1)} = \sigma\left(\left(W^{(1)}\right)^T \mathbf{x}^{(0)} + \boldsymbol{b}^{(1)}\right)$$

- σ is applied element-wise
- $W^{(1)}$ is of size $n_0 \times n_1$
- $\boldsymbol{w}_i^{(1)} \in \mathbb{R}^{n_0}$ columns of $W^{(1)}$
- $\boldsymbol{b}^{(1)} \in \mathbb{R}^{n_1}$ groups all the biases
- The parameters are $\Theta^{(1)} = \{W^{(1)}, \boldsymbol{b}^{(1)}\}$



Two hidden layers

- n_k is the number of neurons at layer k

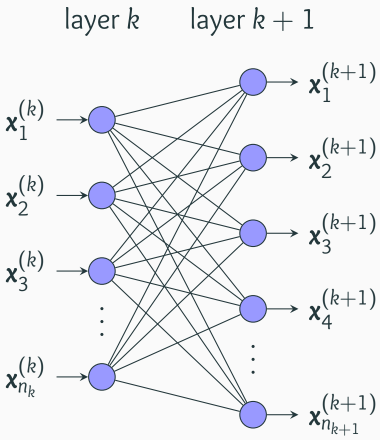
$$\mathbf{x}_i^{(k+1)} = \sigma\left(\sum_{j=1}^{n_k} \mathbf{w}_{ij}^{(k+1)} \mathbf{x}_j^{(k)} + \mathbf{b}_i^{(k+1)}\right)$$

- Condensed version

$$\mathbf{x}^{(k+1)} = \sigma\left(\left(W^{(k+1)}\right)^T \mathbf{x}^{(k)} + \mathbf{b}^{(k+1)}\right)$$

- The parameters at layer $k+1$ are

$$\Theta^{(k+1)} = \{W^{(k+1)}, \mathbf{b}^{(k+1)}\}$$



All layers

- Transformation at layer k:

$$F_{\Theta_k}(\mathbf{x}) = \sigma\left(\left(W^{(k)}\right)^T \mathbf{x} + \mathbf{b}^{(k)}\right)$$

- Suppose the neural network has K layers (input excluded, output included)

$$\begin{aligned}
\mathbf{x}^{(K)} &= F_{\Theta_K}\left(\mathbf{x}^{(K-1)}\right) \\
&\vdots \\
\mathbf{x}^{(K)} &= F_{\Theta_K}\left(F_{\Theta_{K-1}}\left(\dots F_{\Theta_1}\left(\mathbf{x}^{(0)}\right)\right)\right) = F_{\Theta}\left(\mathbf{x}^{(0)}\right)
\end{aligned}$$

- Parameters are

$$\Theta = \left\{\mathbf{b}^{(1)}, W^{(1)}, \mathbf{b}^{(2)}, W^{(2)}, \dots, \mathbf{b}^{(K)}, W^{(K)}\right\}$$

Loss function

- Measure the discrepancy between
    - the prediction $F_\Theta(\boldsymbol{x})$
    - the expected output $\boldsymbol{y} = (y_1, \ldots, y_{n_K})$
- Denoted $\ell(F_\Theta(\boldsymbol{x}), \boldsymbol{y})$
- Output $\boldsymbol{x} = (x_1, \ldots, x_{n_K})$
- Expected output $\boldsymbol{y} = (y_1, \ldots, y_{n_K})$
- Mean squared error (MSE) loss:

$$\frac{1}{n_K} \|\boldsymbol{x} - \boldsymbol{y}\|^2 = \frac{1}{n_K} \sum_{i=1}^{n_K} (x_i - y_i)^2$$

Cross entropy

- Adapted to expected output in range $(0, 1)$
- In classification task, all the y_i's are *zero* except one (one-hot encoding)
- The output **has to** take values in that range
- Use normalizing transform if necessary (softmax)
- Cross entropy error

$$\mathrm{CE}(\boldsymbol{x}, \boldsymbol{y}) = -\sum_{i=1}^{n_K} y_i \log x_i$$

- If $\boldsymbol{y}$ is one-hot encoded, $\mathrm{CE}(\boldsymbol{x}, \boldsymbol{y}) = -\log x_k$ with $y_k = 1$

Softmax

- Parameter-free normalizing transform

$$\text{softmax} : \mathbb{R}^{n_K} \to \{(p_1, \ldots, p_{n_K}) \in \mathbb{R}^{n_K},\ p_i \geqslant 0, p_1 + \cdots + p_{n_K} = 1\}$$

- If $\boldsymbol{z} = \text{softmax}\left(\boldsymbol{x}\right)$ we have

$$z_i = \frac{\exp x_i}{\sum_{j=1}^{n_K} \exp x_j}$$

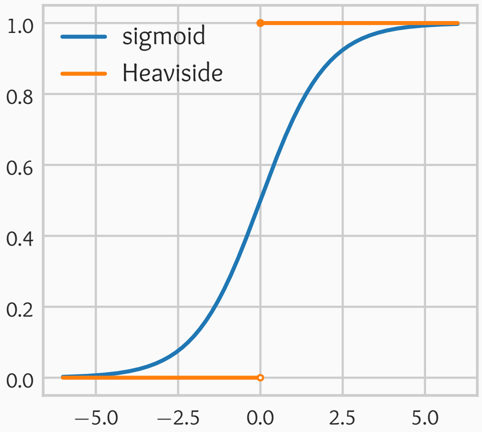
- $\text{softmax}\left(\boldsymbol{x} + c\mathbb{1}\right) = \text{softmax}\left(\boldsymbol{x}\right)$, with $c \in \mathbb{R}$
- If $\boldsymbol{x}_i \geqslant \boldsymbol{x}_j$ then $\text{softmax}\left(\boldsymbol{x}\right)_i \geqslant \text{softmax}\left(\boldsymbol{x}\right)_j$
- Really an "arg softmax" rather than a "softmax"

Activation functions

Logistic function

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

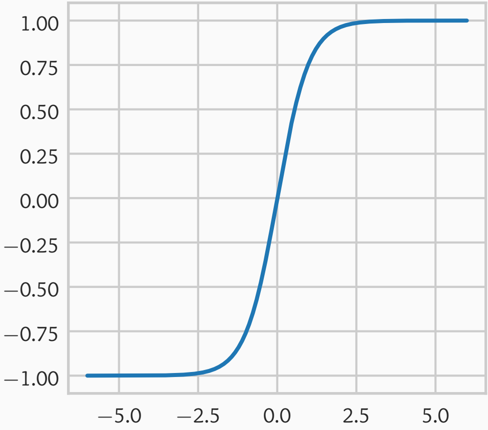
- Saturates at 0 and 1 when $x \to \pm\infty$
- Smooth version of Heaviside function
- $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- Killing gradients



Hyperbolic tangent

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

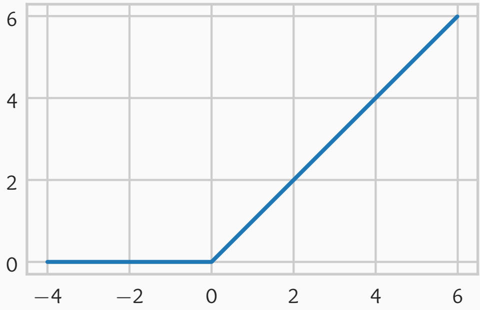
- Symmetric sigmoid
- Saturates at ± 1 when $x \to \pm\infty$
- $\tanh'(x) = 1 + \tanh^2(x)$
- Linearly related to the sigmoid function by
$$\tanh(x) = 2\sigma(2x) - 1$$
- Killing gradients



Rectified linear unit (ReLU)

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

- Learns faster that sigmoid-like activation function
- Simpler to compute
- Provide sparsity of activations
- Dead ReLU: never activated across whole training set.
- Non negative activation function: zig-zag learning (equation (4))

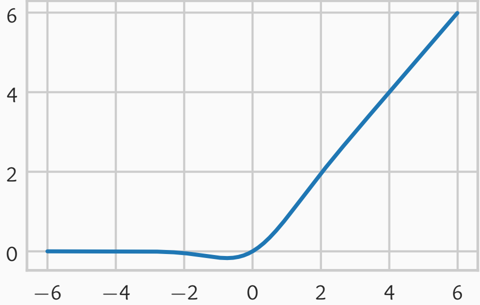


Gaussian Error Linear Unit (GELU), see Hendrycks and Gimpel 2023

$$\mathrm{GELU}(x) = x\Phi(x)$$

where Φ is the standard Gaussian cumulative distribution function

- Smoother version of ReLU
- Mostly used in transformers



Summary

- Use ReLU
- Sigmoid not used anymore
- Prefer hyperbolic tangent to sigmoid

Gradient descent algorithms

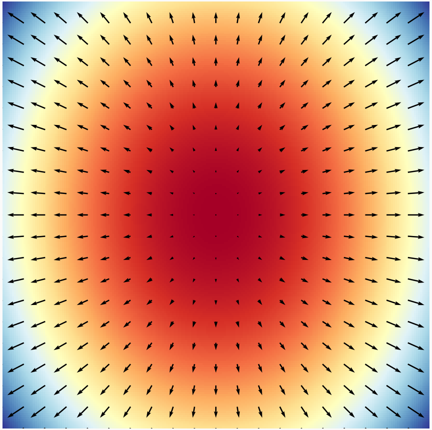
Gradient of a (scalar) function

- Differentiable function

$$f(x,y) = 1.5x^2 + y^2 + 2$$

- The gradient

$$\nabla_{(x,y)} f = \begin{pmatrix} 3x \\ 2y \end{pmatrix}$$



Gradient descent: motivation I

Gradient descent step improves current solution:

- Suppose $\mathcal{L}$ is a differentiable function we want to minimize

$$\arg\min_{\boldsymbol{\theta}\in\Theta} \mathcal{L}(\boldsymbol{\theta})$$

- Starting from the first order Taylor expansion. For a small $\|\mathbf{h}\|$ we have

$$\mathcal{L}(\boldsymbol{\theta}+\mathbf{h}) \approx \mathcal{L}(\boldsymbol{\theta}) + \langle \nabla_{\boldsymbol{\theta}}\mathcal{L}, \mathbf{h}\rangle$$

- Choose $\mathbf{h} = -\eta\nabla_{\boldsymbol{\theta}}\mathcal{L}$ (the gradient descent step), we have

$$\mathcal{L}(\boldsymbol{\theta}-\eta\nabla_{\boldsymbol{\theta}}\mathcal{L}) \approx \mathcal{L}(\boldsymbol{\theta}) - \eta\left\|\nabla_{\boldsymbol{\theta}}\mathcal{L}\right\|^2 < \mathcal{L}(\boldsymbol{\theta})$$

- $\boldsymbol{\theta}-\eta\nabla_{\boldsymbol{\theta}}\mathcal{L}$ is better than $\boldsymbol{\theta}$

Gradient descent: motivation II

Gradient descent step yields best update of linearized and regularized objective function

- Looking for the best $\boldsymbol{\theta}' = \boldsymbol{\theta} + \mathbf{h}$ around fixed $\boldsymbol{\theta}$
- Instead of minimizing $\mathcal{L}(\boldsymbol{\theta}')$, we minimize

$$\mathcal{L}(\boldsymbol{\theta}') \approx \mathcal{L}(\boldsymbol{\theta}) + \langle \mathbf{h}, \nabla_{\boldsymbol{\theta}} \mathcal{L} \rangle \tag{1}$$

- $\boldsymbol{\theta}'$ should stay close to $\boldsymbol{\theta}$ for (1) to hold, we penalize by $\left\|\boldsymbol{\theta} - \boldsymbol{\theta}'\right\| = \mathbf{h}$

$$\mathcal{L}(\boldsymbol{\theta}) + \langle \mathbf{h}, \nabla_{\boldsymbol{\theta}} \mathcal{L} \rangle + \frac{1}{2\eta} \|\mathbf{h}\|^2$$

- Minimizing w.r.t $\mathbf{h}$ gives

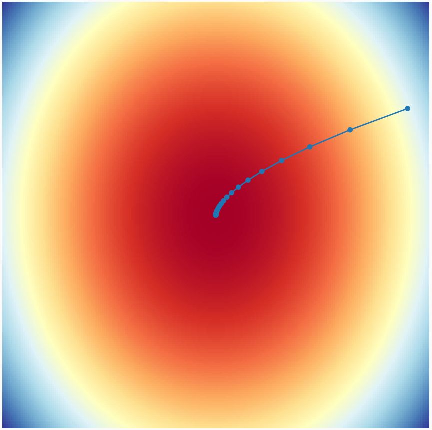
$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}$$

Gradient descent algorithm

- Starting point $\boldsymbol{\theta}_0$
- Gradient descent step

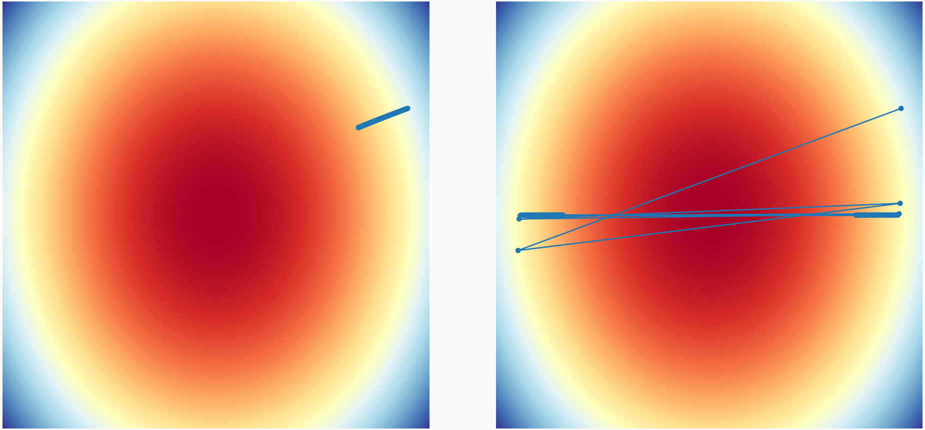
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}_t} \mathcal{L}$$

- η is the *learning rate*
- Learning rate too small: slow convergence
- Learning rate too high: fluctuate around minimum or even diverge



(a) $\eta = 0.1$

Gradient descent: illustration



(a) $\eta = 0.001$ (b) $\eta = 0.665757$

Gradient descent for learning

- Empirical risk minimization (ERM)

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \ell(F_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i)$$

- Computing $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$ requires computing $\nabla_{\boldsymbol{\theta}}\ell(F_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i)$ for all $i = 1, \ldots, n$!
- Efficient when $n < 10000$
- Impractical when $n > 10000$

Stochastic gradient descent (SGD)

- Directly minimizing the expected risk

$$\mathcal{L}_{\mathcal{D}}(\boldsymbol{\theta}) = \mathop{\mathbb{E}}_{\mathbf{x},y\sim\mathcal{D}} \ell(F_{\boldsymbol{\theta}}(\mathbf{x}), y)$$

- Applying the gradient operator $\nabla_{\boldsymbol{\theta}}$ yields

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\mathcal{D}}(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \mathop{\mathbb{E}}_{\mathbf{x},y\sim\mathcal{D}} \ell(F_{\boldsymbol{\theta}}(\mathbf{x}), y) \\ &= \mathop{\mathbb{E}}_{\mathbf{x},y\sim\mathcal{D}} \nabla_{\boldsymbol{\theta}}\ell(F_{\boldsymbol{\theta}}(\mathbf{x}), y)\end{aligned}$$

- $\nabla_{\boldsymbol{\theta}}\ell(F_{\boldsymbol{\theta}}(\mathbf{x}), y)$ is an unbiased estimator of $\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\mathcal{D}}(\boldsymbol{\theta})$

Minibatch Stochastic gradient descent

- Easy to get an observation of $\nabla_{\boldsymbol{\theta}} \ell(F_{\boldsymbol{\theta}}(\mathbf{x}), y)$
- Unbiased but possibly of high variance
- Combining unbiased estimator to reduce variance

$$\mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|} \sum_{(\mathbf{x}, y) \in \mathcal{B}} \ell(F_{\boldsymbol{\theta}}(\mathbf{x}), y)$$

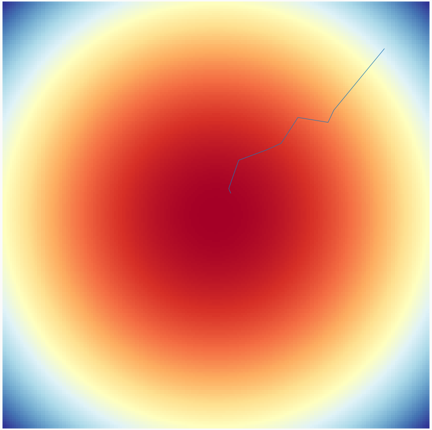
- $\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}}$ is an unbiased estimate of $\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{D}}$
- $\mathbb{E}_{\mathcal{B}} \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}} = \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{D}}$
- Size of minibatch $|\mathcal{B}|$ is a tradeoff term between good estimate of the gradient and cheap computations

- Model $Y = \langle \boldsymbol{w}_0, X \rangle + \varepsilon$ with:
  - $X \sim \mathcal{N}(0, \Sigma)$
  - $\varepsilon \sim \mathcal{N}\left(0, \sigma^2\right)$
- Minimization problem

$$\arg\min_{\boldsymbol{w} \in \mathbb{R}^p} \mathbb{E}\left(Y - \langle \boldsymbol{w}, X \rangle\right)^2$$

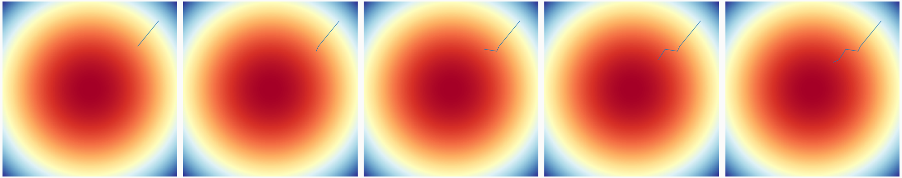
- One can show that

$$\mathbb{E}\left(Y - \langle \boldsymbol{w}, X \rangle\right)^2 = (\boldsymbol{w} - \boldsymbol{w}_0)^T \Sigma (\boldsymbol{w} - \boldsymbol{w}_0)$$

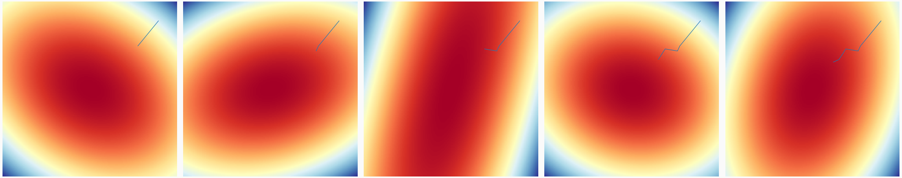


Stochastic gradient descent path

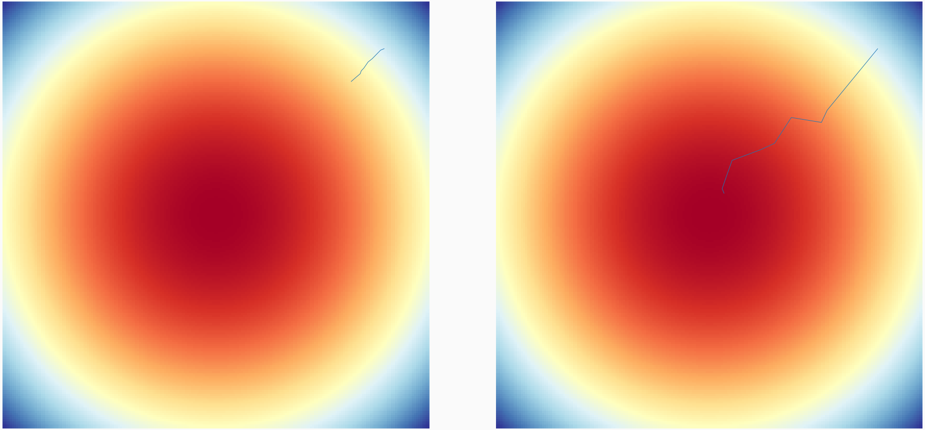
- Gradient descent for first minibatches, theoretical loss is $\mathbb{E}\left(Y - \langle \boldsymbol{w}, X \rangle\right)^2$



- What SGD is really seeing through minibatches, loss is $\dfrac{1}{|\mathcal{B}|} \displaystyle\sum_{(\mathbf{x},y)\in\mathcal{B}} \left(y - \langle \boldsymbol{w}, \mathbf{x} \rangle\right)^2$

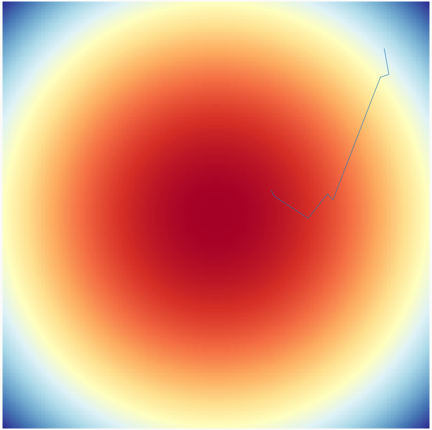


Influence of learning rate

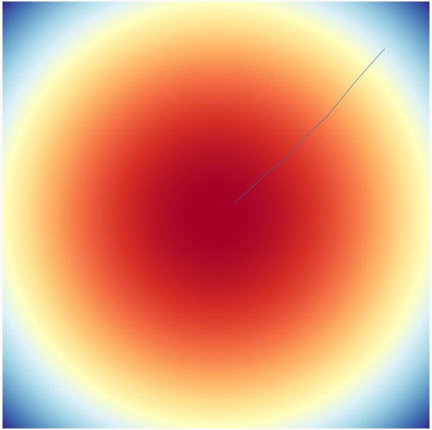


(a) $\eta = 0.01$, $|\mathcal{B}| = 10$ (b) $\eta = 0.1$, $|\mathcal{B}| = 10$

Influence of minibatch size



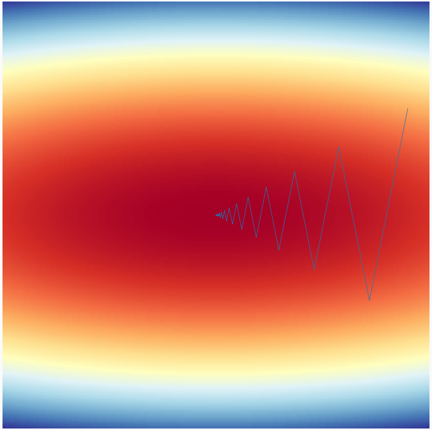
(a) $\eta = 0.1$, $|\mathcal{B}| = 1$



(b) $\eta = 0.1$, $|\mathcal{B}| = 100$

Gradient descent with momentum

- If learning rate is high or loss landscape is bumpy SGD (and GD) trajectory might be erratic



(a) $\eta = 0.1$

Gradient descent with momentum

- Original update rule

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}_t} \mathcal{L}$$

- Gradient descent rule with momentum ($\mathbf{v}_0 = \nabla_{\boldsymbol{\theta}_0} \mathcal{L}$)

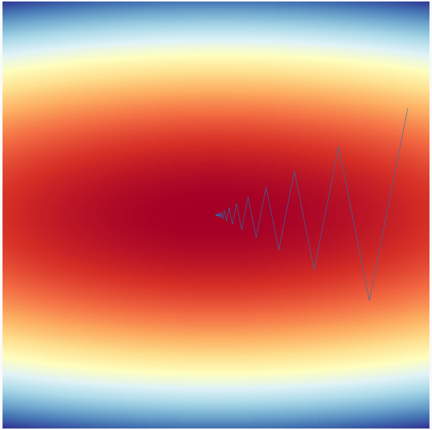
$$\begin{aligned} \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \mathbf{v}_t \\ \mathbf{v}_t &= \beta \mathbf{v}_{t-1} + (1-\beta) \nabla_{\boldsymbol{\theta}_t} \mathcal{L} \end{aligned}$$

- Average current gradient with previous one
- Exponentially weighted moving average on past gradients

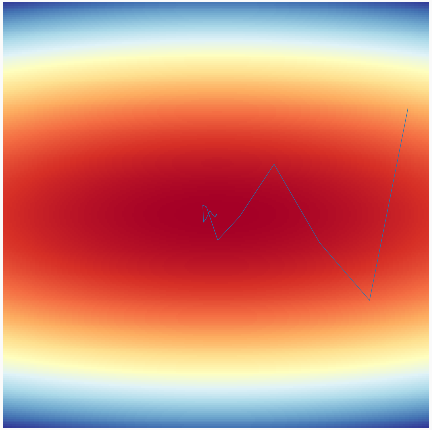
$$\mathbf{v}_t = (1-\beta) \sum_{i=0}^{t-1} \beta^i \nabla_{\boldsymbol{\theta}_{t-i}} \mathcal{L} + \beta^t \nabla_{\boldsymbol{\theta}_0} \mathcal{L}$$

- Averaging so better estimate

Momentum in action (GD)



(a) $\eta = 0.1$, no momentum. Gradients are abruptly changing



(b) $\eta = 0.1$, momentum $= 0.5$. Gradients change is smoothed out

See https://distill.pub/2017/momentum/

Other stochastic optimizers

- Adagrad, RMSProp, **Adam**: decreases the learning rate dynamically coordinate-wise
- Adadelta: adapt learning rate from rate of change in the parameters

Backpropagation algorithm

Backpropagation

- Chain rule

$$(f \circ g)'(\theta) = \big(f' \circ g\big)(\theta) \cdot g'(\theta)$$

- Generalization to any number of functions ($F = f_3 \circ f_2 \circ f1$)

$$\begin{aligned} F'(\theta) = (f_3 \circ f_2 \circ f_1)'(\theta) &= \big(f_3' \circ f_2 \circ f_1\big)(\theta) \cdot \big(f_2' \circ f_1\big)(\theta) \cdot f_1'(\theta) \\ &= f_3'((f_2 \circ f_1)(\theta)) \cdot f_2'(f_1(\theta)) \cdot f_1'(\theta) \end{aligned}$$

Backpropagation

- Chain rule

$$(f \circ g)'(\theta) = \big(f' \circ g\big)(\theta) \cdot g'(\theta)$$

- Generalization to any number of functions ($F = f_3 \circ f_2 \circ f1$)

$$\begin{aligned} F'(\theta) = (f_3 \circ f_2 \circ f_1)'(\theta) &= \big(f_3' \circ f_2 \circ f_1\big)(\theta) \cdot \big(f_2' \circ f_1\big)(\theta) \cdot f_1'(\theta) \\ &= f_3'((f_2 \circ f_1)(\theta)) \cdot f_2'(f_1(\theta)) \cdot f_1'(\theta) \end{aligned}$$

- Forward and backward pass

$$\theta \xrightarrow{\quad f_1 \quad} f_1(\theta)$$

Backpropagation

- Chain rule

$$(f \circ g)'(\theta) = \big(f' \circ g\big)(\theta) \cdot g'(\theta)$$

- Generalization to any number of functions ($F = f_3 \circ f_2 \circ f1$)

$$\begin{aligned} F'(\theta) = (f_3 \circ f_2 \circ f_1)'(\theta) &= \big(f_3' \circ f_2 \circ f_1\big)(\theta) \cdot \big(f_2' \circ f_1\big)(\theta) \cdot f_1'(\theta) \\ &= f_3'((f_2 \circ f_1)(\theta)) \cdot f_2'(f_1(\theta)) \cdot f_1'(\theta) \end{aligned}$$

- Forward and backward pass

$$\theta \xrightarrow{\quad f_1 \quad} f_1(\theta) \xrightarrow{\ f_2\ } f_2(f_1(\theta))$$

Backpropagation

- Chain rule

$$(f \circ g)'(\theta) = \big(f' \circ g\big)(\theta) \cdot g'(\theta)$$

- Generalization to any number of functions ($F = f_3 \circ f_2 \circ f1$)

$$\begin{aligned} F'(\theta) = (f_3 \circ f_2 \circ f_1)'(\theta) &= \big(f_3' \circ f_2 \circ f_1\big)(\theta) \cdot \big(f_2' \circ f_1\big)(\theta) \cdot f_1'(\theta) \\ &= f_3'((f_2 \circ f_1)(\theta)) \cdot f_2'(f_1(\theta)) \cdot f_1'(\theta) \end{aligned}$$

- Forward and backward pass

$$\theta \xrightarrow{\quad f_1 \quad} f_1(\theta) \xrightarrow{\quad f_2 \quad} f_2(f_1(\theta)) \xrightarrow{f_3} f_3(f_2(f_1(\theta))) = F(\theta)$$

Backpropagation

- Chain rule

$$(f \circ g)'(\theta) = \big(f' \circ g\big)(\theta) \cdot g'(\theta)$$

- Generalization to any number of functions ($F = f_3 \circ f_2 \circ f1$)

$$\begin{aligned} F'(\theta) = (f_3 \circ f_2 \circ f_1)'(\theta) &= \big(f_3' \circ f_2 \circ f_1\big)(\theta) \cdot \big(f_2' \circ f_1\big)(\theta) \cdot f_1'(\theta) \\ &= f_3'((f_2 \circ f_1)(\theta)) \cdot f_2'(f_1(\theta)) \cdot f_1'(\theta) \end{aligned}$$

- Forward and backward pass

$$\begin{tikzcd} \theta \arrow[r, "f_1"] & f_1(\theta) \arrow[r, "f_2"] & f_2(f_1(\theta)) \arrow[r, "f_3"] \arrow[d] & f_3(f_2(f_1(\theta))) = F(\theta) \\ & & f_3'(f_2(f_1(\theta))) & \end{tikzcd}$$

Backpropagation

- Chain rule

$$(f \circ g)'(\theta) = \big(f' \circ g\big)(\theta) \cdot g'(\theta)$$

- Generalization to any number of functions ($F = f_3 \circ f_2 \circ f1$)

$$\begin{aligned} F'(\theta) = (f_3 \circ f_2 \circ f_1)'(\theta) &= \big(f_3' \circ f_2 \circ f_1\big)(\theta) \cdot \big(f_2' \circ f_1\big)(\theta) \cdot f_1'(\theta) \\ &= f_3'((f_2 \circ f_1)(\theta)) \cdot f_2'(f_1(\theta)) \cdot f_1'(\theta) \end{aligned}$$

- Forward and backward pass

$$\begin{tikzcd} \theta \arrow[r, "f_1"] & f_1(\theta) \arrow[r, "f_2"] \arrow[d] & f_2(f_1(\theta)) \arrow[r, "f_3"] \arrow[d] & f_3(f_2(f_1(\theta))) = F(\theta) \\ & f_2'(f_1(\theta)) & f_3'(f_2(f_1(\theta))) \arrow[l, "\times"'] & \end{tikzcd}$$

- Chain rule

$$(f \circ g)'(\theta) = \big(f' \circ g\big)(\theta) \cdot g'(\theta)$$

- Generalization to any number of functions ($F = f_3 \circ f_2 \circ f 1$)

$$\begin{aligned} F'(\theta) = (f_3 \circ f_2 \circ f_1)'(\theta) &= \big(f_3' \circ f_2 \circ f_1\big)(\theta) \cdot \big(f_2' \circ f_1\big)(\theta) \cdot f_1'(\theta) \\ &= f_3'((f_2 \circ f_1)(\theta)) \cdot f_2'(f_1(\theta)) \cdot f_1'(\theta) \end{aligned}$$

- Forward and backward pass

$$\begin{tikzcd} \theta \arrow[r, "f_1"] \arrow[d] & f_1(\theta) \arrow[r, "f_2"] \arrow[d] & f_2(f_1(\theta)) \arrow[r, "f_3"] \arrow[d] & f_3(f_2(f_1(\theta))) = F(\theta) \\ F'(\theta) = f_1'(\theta) & f_2'(f_1(\theta)) \arrow[l, "\times"'] & f_3'(f_2(f_1(\theta))) \arrow[l, "\times"'] & \end{tikzcd}$$

Backpropagation

- Generalizable to $\mathbb{R}^n$ to $\mathbb{R}^p$ functions using jacobians
- Generalizable to a computational graph (DAG)

Backpropagating the loss

Total loss on the minibatch $\mathcal{B}$

$$\mathcal{L}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{(\mathbf{x},\boldsymbol{y}) \in \mathcal{B}} \ell(F_{\Theta}(\mathbf{x}), \boldsymbol{y})$$

Differentiating w.r.t any scalar parameter θ (any $\boldsymbol{w}_{ij}^{(k)}$ or $\boldsymbol{b}_{j}^{(k)}$)

$$\frac{\partial \mathcal{L}_{\mathcal{B}}}{\partial \theta} = \frac{1}{|\mathcal{B}|} \sum_{(\mathbf{x},\boldsymbol{y}) \in \mathcal{B}} \frac{\partial \ell\left(\mathbf{x}^{(K)}, \boldsymbol{y}\right)}{\partial \theta} \quad \text{avec} \quad \mathbf{x}^{(K)} = F_{\Theta}(\mathbf{x})$$

$$\ell\left(\mathbf{x}^{(K)}, \boldsymbol{y}\right) = \frac{1}{n_K} \sum_{i=1}^{n_K} \ell\left(\mathbf{x}_i^{(K)}, \boldsymbol{y}_i\right)$$

It suffices to compute

$$\frac{\partial \ell\left(\mathbf{x}_i^{(K)}, \boldsymbol{y}_i\right)}{\partial \theta}$$

MSE loss

- MSE loss is defined by

$$\ell\left(\boldsymbol{x}_i^{(K)}, \boldsymbol{y}_i\right) = \left(\boldsymbol{x}_i^{(K)} - \boldsymbol{y}_i\right)^2$$

- Differentiating with respect to some scalar parameter $\theta = \boldsymbol{b}_q^{(k)}$ or $\theta = \boldsymbol{w}_{pq}^{(k)}$
- ($\boldsymbol{x}_i^{(K)}$ and $\boldsymbol{y}_i$ are scalars)

$$\begin{aligned}\frac{\partial \ell\left(\boldsymbol{x}_i^{(K)}, \boldsymbol{y}_i\right)}{\partial \theta} &= \frac{\partial \ell\left(\boldsymbol{x}_i^{(K)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{x}_i^{(K)}} \cdot \frac{\partial \boldsymbol{x}_i^{(K)}}{\partial \theta} \\ &= 2\left(\boldsymbol{x}_i^{(K)} - \boldsymbol{y}_i\right) \cdot \frac{\partial \boldsymbol{x}_i^{(K)}}{\partial \theta}\end{aligned}$$

- We need to know $\frac{\partial \boldsymbol{x}_i^{(K)}}{\partial \theta}$ now!

Cross entropy loss

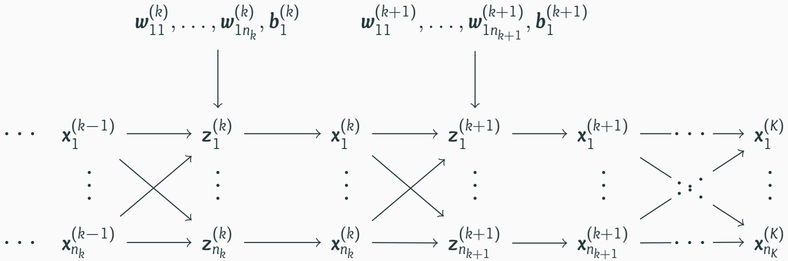
- Cross entropy loss: $\ell\left(\mathbf{x}_i^{(K)}, \boldsymbol{y}_i\right) = -\boldsymbol{y}_i \log \mathbf{x}_i^{(K)}$
- Differentiating with respect to some scalar parameter $\theta = \boldsymbol{b}_q^{(k)}$ or $\theta = \boldsymbol{w}_{pq}^{(k)}$

$$\frac{\partial \ell\left(\mathbf{x}_i^{(K)}, \boldsymbol{y}_i\right)}{\partial \theta} = \frac{\partial \ell\left(\mathbf{x}_i^{(K)}, \boldsymbol{y}_i\right)}{\partial \mathbf{x}_i^{(K)}} \cdot \frac{\partial \mathbf{x}_i^{(K)}}{\partial \theta}$$
$$= -\frac{\boldsymbol{y}_i}{\mathbf{x}_i^{(K)}} \cdot \frac{\partial \mathbf{x}_i^{(K)}}{\partial \theta}$$

- We need to know $\frac{\partial \mathbf{x}_i^{(K)}}{\partial \theta}$ now!

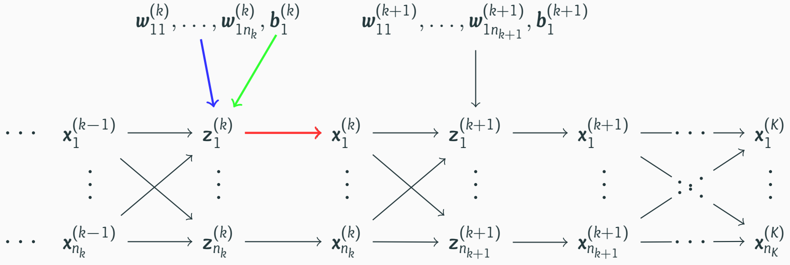
Computational graph

- Define $\mathbf{z}_i^{(k)} = \left\langle \mathbf{w}_i^{(k)}, \mathbf{x}^{(k-1)} \right\rangle + \mathbf{b}_i^{(k)}$ so that we have $\mathbf{x}_i^{(k)} = \sigma\left(\mathbf{z}_i^{(k)}\right)$
- Computational graph for $\mathbf{x}_1^{(k-1)}, \mathbf{x}_1^{(k)}, \mathbf{x}_1^{(k+1)}$ only!



Gradient of last layer w.r.t parameters I

Computational graph for neuron 1 at layer k ($p = 1$):



$$\frac{\partial \mathbf{x}_i^{(K)}}{\partial w_{pq}^{(k)}} = \frac{\partial \mathbf{x}_i^{(K)}}{\partial \mathbf{x}_p^{(k)}} \cdot \frac{\partial \mathbf{x}_p^{(k)}}{\partial \mathbf{z}_p^{(k)}} \cdot \frac{\partial \mathbf{z}_p^{(k)}}{\partial w_{pq}^{(k)}} \qquad \frac{\partial \mathbf{x}_i^{(K)}}{\partial b_p^{(k)}} = \frac{\partial \mathbf{x}_i^{(K)}}{\partial \mathbf{x}_p^{(k)}} \cdot \frac{\partial \mathbf{x}_p^{(k)}}{\partial \mathbf{z}_p^{(k)}} \cdot \frac{\partial \mathbf{z}_p^{(k)}}{\partial b_p^{(k)}}$$

Gradient of last layer w.r.t parameters II

Given that $\mathbf{x}_p^{(k)} = \sigma\left(\mathbf{z}_p^{(k)}\right)$, we have $\boxed{\dfrac{\partial \mathbf{x}_p^{(k)}}{\partial \mathbf{z}_p^{(k)}}} = \sigma'\left(\mathbf{z}_p^{(k)}\right)$

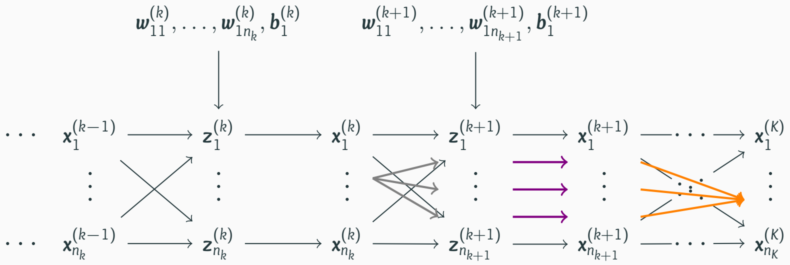
And $\mathbf{z}_p^{(k)} = \left\langle \boldsymbol{w}_p^{(k)}, \mathbf{x}^{(k-1)} \right\rangle + \boldsymbol{b}_p^{(k)}$, so $\boxed{\dfrac{\partial \mathbf{z}_p^{(k)}}{\partial \boldsymbol{w}_{pq}^{(k)}}} = \mathbf{x}_q^{(k-1)}$ and $\boxed{\dfrac{\partial \mathbf{z}_p^{(k)}}{\partial \boldsymbol{b}_p^{(k)}} = 1}$

Finally

$$\boxed{\frac{\partial \mathbf{x}_i^{(K)}}{\partial \boldsymbol{w}_{pq}^{(k)}} = \frac{\partial \mathbf{x}_i^{(K)}}{\partial \mathbf{x}_p^{(k)}} \cdot \sigma'\left(\mathbf{z}_p^{(k)}\right) \cdot \mathbf{x}_q^{(k-1)}} \qquad \boxed{\frac{\partial \mathbf{x}_i^{(K)}}{\partial \boldsymbol{b}_p^{(k)}} = \frac{\partial \mathbf{x}_i^{(K)}}{\partial \mathbf{x}_p^{(k)}} \cdot \sigma'\left(\mathbf{z}_p^{(k)}\right)}$$

Need to compute $\dfrac{\partial \mathbf{x}_i^{(K)}}{\partial \mathbf{x}_p^{(k)}}$ now!

Gradient of last layer w.r.t other layer



$$\frac{\partial \mathbf{x}_i^{(K)}}{\partial \mathbf{x}_p^{(k)}} = \sum_{j=1}^{n_{k+1}} \frac{\partial \mathbf{x}_i^{(K)}}{\partial \mathbf{x}_j^{(k+1)}} \cdot \frac{\partial \mathbf{x}_j^{(k+1)}}{\partial \mathbf{z}_j^{(k+1)}} \cdot \frac{\partial \mathbf{z}_j^{(k+1)}}{\partial \mathbf{x}_p^{(k)}}$$

Backpropagating from next layer

Given that $\mathbf{x}_j^{(k+1)} = \sigma\left(\mathbf{z}_j^{(k+1)}\right)$, we have

$$\boxed{\frac{\partial \mathbf{x}_j^{(k+1)}}{\partial \mathbf{z}_j^{(k+1)}}} = \sigma'\left(\mathbf{z}_j^{(k+1)}\right)$$

And $\mathbf{z}_j^{(k+1)} = \left\langle \boldsymbol{w}_j^{(k+1)}, \mathbf{x}^{(k)} \right\rangle + \boldsymbol{b}_j^{(k+1)}$, so

$$\boxed{\frac{\partial \mathbf{z}_j^{(k+1)}}{\partial \mathbf{x}_p^{(k)}}} = \boldsymbol{w}_{jp}^{(k+1)}$$

Replacing in last equation, we have

$$\boxed{\frac{\partial \mathbf{x}_i^{(K)}}{\partial \mathbf{x}_p^{(k)}} = \sum_{j=1}^{n_{k+1}} \frac{\partial \mathbf{x}_i^{(K)}}{\partial \mathbf{x}_j^{(k+1)}} \cdot \sigma'\left(\mathbf{z}_j^{(k+1)}\right) \cdot \boldsymbol{w}_{jp}^{(k+1)}}$$

Backpropagation equations

- Backpropagation equations for parameters $\boldsymbol{w}_{pq}^{(k)}$ and $\boldsymbol{b}_p^{(k)}$

$$\frac{\partial \mathbf{x}_i^{(K)}}{\partial \boldsymbol{w}_{pq}^{(k)}} = \frac{\partial \mathbf{x}_i^{(K)}}{\partial \mathbf{x}_p^{(k)}} \cdot \sigma'\left(\mathbf{z}_p^{(k)}\right) \cdot \mathbf{x}_q^{(k-1)} \tag{2}$$

$$\frac{\partial \mathbf{x}_i^{(K)}}{\partial \boldsymbol{b}_p^{(k)}} = \frac{\partial \mathbf{x}_i^{(K)}}{\partial \mathbf{x}_p^{(k)}} \cdot \sigma'\left(\mathbf{z}_p^{(k)}\right) \tag{3}$$

- Backpropagation equation

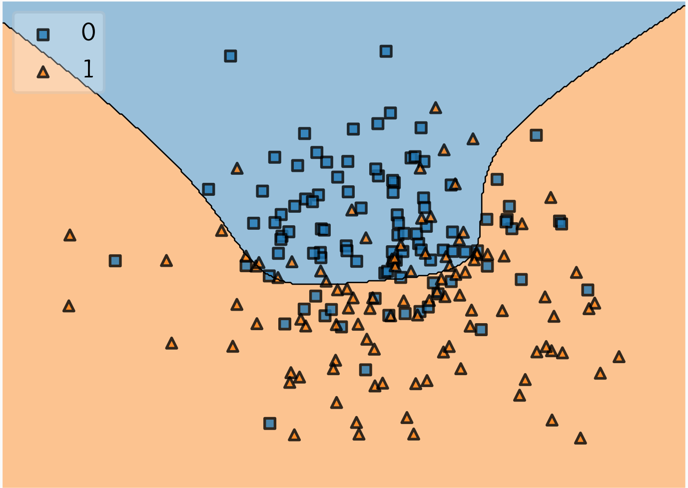
$$\frac{\partial \mathbf{x}_i^{(K)}}{\partial \mathbf{x}_p^{(k)}} = \sum_{j=1}^{n_{k+1}} \frac{\partial \mathbf{x}_i^{(K)}}{\partial \mathbf{x}_j^{(k+1)}} \cdot \sigma'\left(\mathbf{z}_j^{(k+1)}\right) \cdot \boldsymbol{w}_{jp}^{(k+1)} \tag{4}$$

- Only one sweep backward is necessary to compute all gradients

Regularization

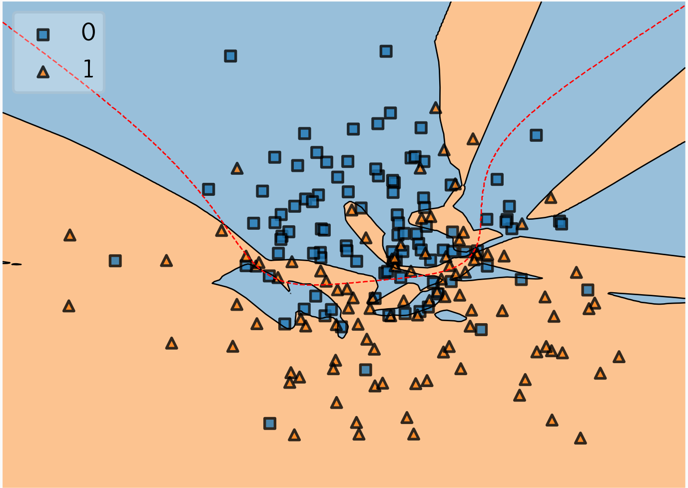
Toy dataset

- 200 samples, 2 classes
- Gaussian mixture model
- Bayes decision boundary



Standard neural network classification

- Neural network
    - 1 hidden layer with 50 units
    - $\sim$ 200 parameters
- SGD algorithm
    - learning rate: 0.1
    - momentum: 0.9



Regularizing the ERM

- Empirical risk minimization (ERM)

$$\arg\min_{\theta\in\Theta} \mathcal{L}_{\mathcal{B}} = \arg\min_{\theta\in\Theta} \frac{1}{|\mathcal{B}|} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{B}} \ell(F_{\Theta}(\mathbf{x}),\mathbf{y})$$

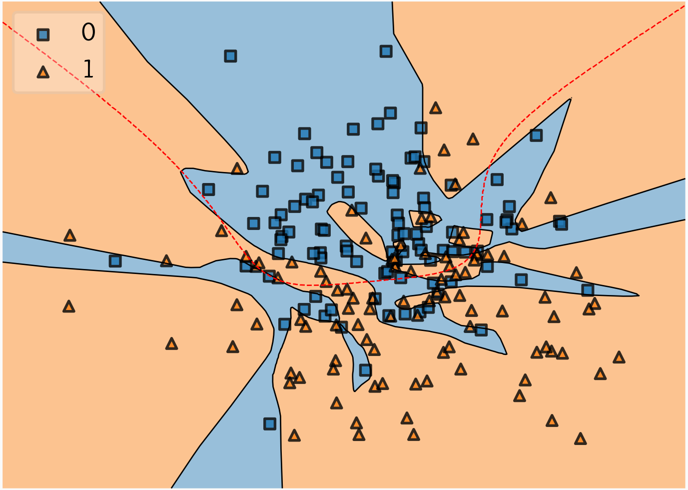
- L_2 penalizing term

$$\arg\min_{\theta\in\Theta} \mathcal{L}_{\mathcal{R}} = \arg\min_{\theta\in\Theta} \frac{1}{|\mathcal{B}|} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{B}} \ell(F_{\Theta}(\mathbf{x}),\mathbf{y}) + \lambda \sum_{k=1}^{K} \left\| W^{(k)} \right\|_F$$

- Biases terms are not regularized
- λ is the tradeoff parameter called *weight decay*

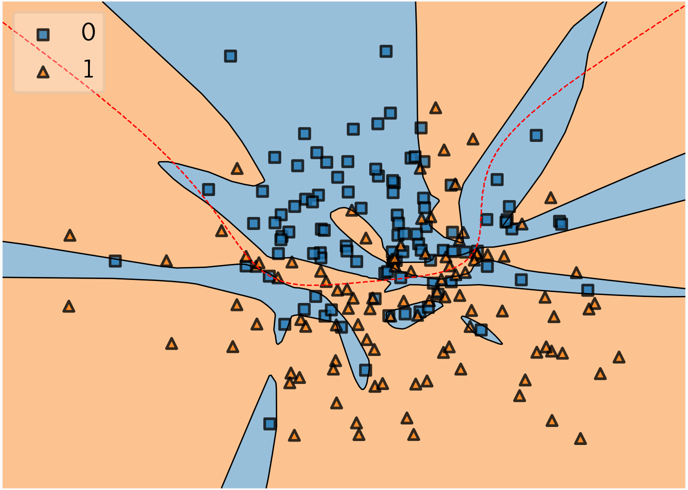
Weight decay: an example

- SGD algorithm
  - learning rate: 0.1
  - momentum: 0.9
- **weight decay:** 10^{-4}



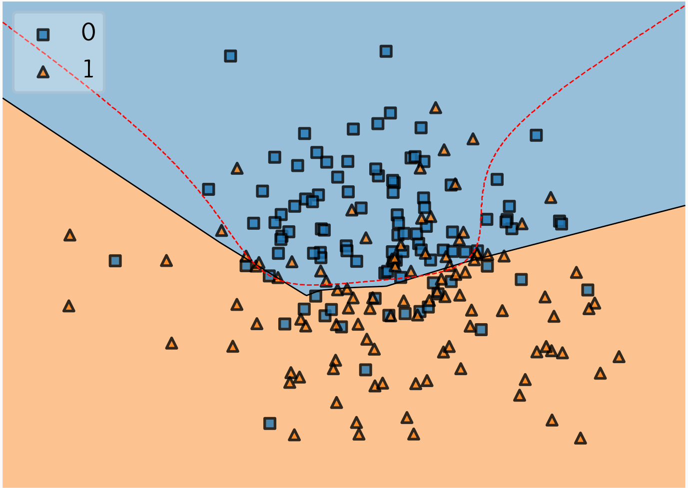
Weight decay: an example

- SGD algorithm
    - learning rate: 0.1
    - momentum: 0.9
- **weight decay:** 10^{-3}



Weight decay: an example

- SGD algorithm
  - learning rate: 0.1
  - momentum: 0.9
- **weight decay:** 10^{-2}



Weight decay: gradient

- One extra term in the loss

$$\mathcal{L}_{\mathcal{R}} = \mathcal{L}_{\mathcal{B}} + \lambda \sum_{k=1}^{K} \left\| W^{(k)} \right\|_F$$

- Gradient is easy to get

$$\frac{\partial \mathcal{L}_{\mathcal{R}}}{\partial \boldsymbol{w}_{ij}^{(k)}} = \frac{\partial \mathcal{L}_{\mathcal{B}}}{\partial \boldsymbol{w}_{ij}^{(k)}} + 2\lambda \boldsymbol{w}_{ij}^{(k)}$$

- Gradients w.r.t. $\boldsymbol{b}_i^{(k)}$ are unchanged

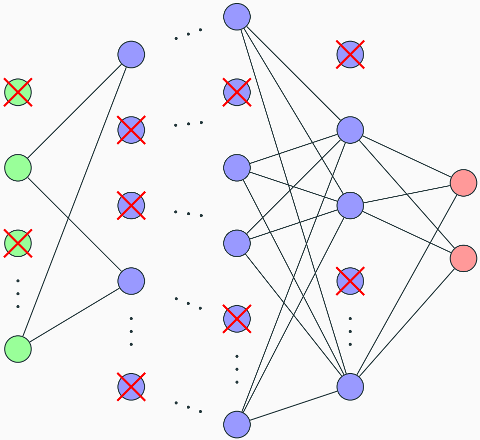
Dropout Srivastava et al. 2014

- Randomly kill nodes in layers during training time

$$\mathbf{x}^{(k)} = \sigma\left(W^{(k)}\left(\mathbf{x}^{(k-1)} \odot \mathbf{h}^{(k)}\right) + \boldsymbol{b}^{(k)}\right)$$

with $\mathbf{h}_k \sim \mathcal{B}(h_k)^{\otimes n_k}$

- h_k is the rate of dropout at layer k
- Rescaling needed at test-time



Batch normalization Ioffe and Szegedy 2015

- Normalize each activation independently from the minibatch statistics

$$\boldsymbol{\mu}^{(k)} = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x}^{(k)} \in \mathcal{B}^{(k)}} \mathbf{z}^{(k)}$$

$$\sigma_i^{(k)} = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x}^{(k)} \in \mathcal{B}^{(k)}} \left(\mathbf{z}_i^{(k)} - \boldsymbol{\mu}_i^{(k)}\right)^2$$

- For each element in the minibatch, replace $\mathbf{z}_i^{(k)}$ and $\mathbf{x}_i^{(k+1)}$ by

$$\tilde{\mathbf{z}}_i^{(k)} = \frac{\mathbf{z}_i^{(k)} - \boldsymbol{\mu}_i^{(k)}}{\sigma_i^{(k)}} \qquad \tilde{\mathbf{x}}_i^{(k+1)} = \sigma\left(\gamma_i^{(k)} \tilde{\mathbf{z}}_i^{(k)} + \beta_i^{(k)}\right)$$

- $\gamma_i^{(k)}$ and $\boldsymbol{\beta}_i^{(k)}$ are $2n_k$ extra parameters
- The $\boldsymbol{b}_i$'s from $\mathbf{z}_i^{(k)} = \left\langle \boldsymbol{w}_i^{(k)}, \mathbf{x}^{(k)} \right\rangle + \boldsymbol{b}_i^{(k)}$ are useless ($\tilde{\mathbf{z}}_i^{(k)}$ does not depend on $\boldsymbol{b}_i$)

References i

[1] Nitish Srivastava et al. **"Dropout: a simple way to prevent neural networks from overfitting."**. In: *Journal of machine learning research* 15.1 (1 2014), pp. 1929–1958.

[2] Sergey Ioffe and Christian Szegedy. **"Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift"**. Feb. 10, 2015. arXiv: `1502.03167 [cs]`. URL: `http://arxiv.org/abs/1502.03167` (visited on 11/06/2017).

[3] Ian Goodfellow et al. ***Deep learning.*** Vol. 1. MIT press Cambridge, 2016.

[4] Aston Zhang et al. ***Dive into deep learning.*** 2020.

[5] Dan Hendrycks and Kevin Gimpel. ***Gaussian Error Linear Units (GELUs).*** June 5, 2023. DOI: `10.48550/arXiv.1606.08415`. arXiv: `1606.08415 [cs]`. URL: `http://arxiv.org/abs/1606.08415` (visited on 11/07/2023). preprint.