Introduction to Deep Learning

Mnacho Echenim

Grenoble INP-Ensimag

2022-2023

Notes

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: $\forall k, p \in \{1 \ldots, n_j\}$, $\zeta^j_k = \zeta^j_p$, hence $\alpha^j_k = \alpha^j_p$ (induction)
- Backpropagation: for all layers j, error components have the same value: $\forall k, p \in \{1 \ldots, n_j\}$, $\mathcal{B}^j_k = \mathcal{B}^j_p = \mathfrak{b}^j$ (backward induction)
- Gradient computation: for all layers j we have

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

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

Notes

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

Notes

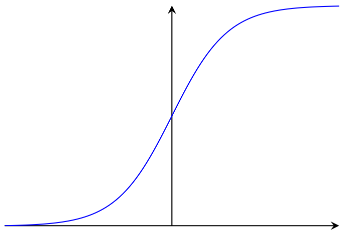
Activation functions

- 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`

Notes

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



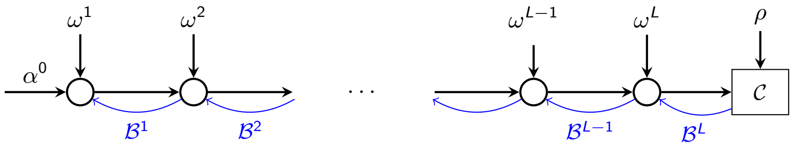
Notes

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 $\mathcal{B}_k^{j+1}$

Notes

Vanishing gradients: illustration



We have:

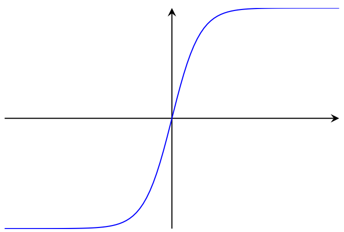
$$\begin{aligned}
\mathcal{B}^L &= \Phi'(\zeta^L)\cdot\mathcal{C}'(\alpha^L,\rho) &\leq \frac{\mathcal{C}'(\alpha^L,\rho)}{4}\\
\mathcal{B}^{L-1} &= \Phi'(\zeta^{L-1})\cdot\omega^L\cdot\mathcal{B}^L &\leq \omega^L\cdot\frac{\mathcal{C}'(\alpha^L,\rho)}{4^2}\\
&\vdots\\
\mathcal{B}^i &= \Phi'(\zeta^i)\cdot\omega^{i+1}\cdot\mathcal{B}^{i+1}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

- 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

Notes

tanh activation function

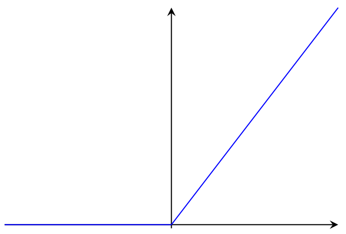
- $\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$
- $\tanh'(x) = 1-(\tanh(x))^2$
- Centered around 0
- There is still the vanishing gradient problem



Notes

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



Notes

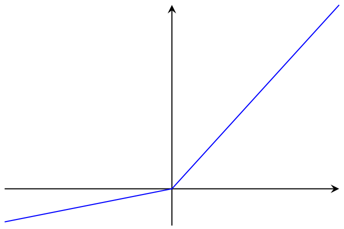
On dead neurons

- 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

Notes

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)



Notes

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

Notes

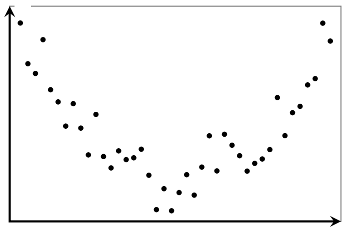
What should the goal of training a neural network be?

- 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**

Notes

What can happen when training a neural network

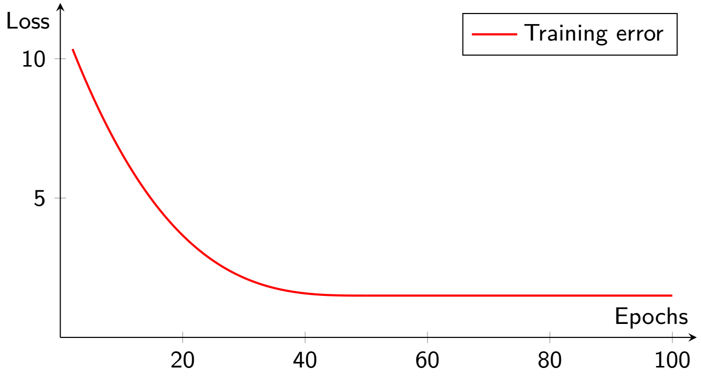
- At the beginning, the neural network is completely off and the training error is large, as the network parameters have not been updated yet (**underfitting**)
- After some time, the training error has been reduced and the network starts behaving nicely
- As we keep training the network, the training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



Notes

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")



Notes

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 (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

Notes

Penalties on weights

Assumption

- 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

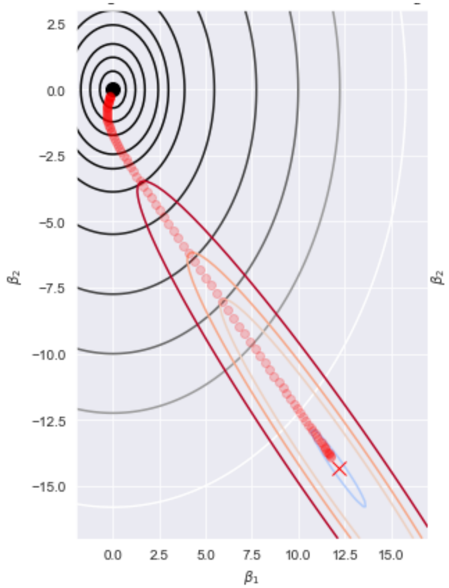
$$\mathcal{E}'(S,\theta) \stackrel{\text{def}}{=} \mathcal{E}(S,\theta) + \kappa \mathcal{P}(\theta)$$

  - 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

Notes

L^2 regularization

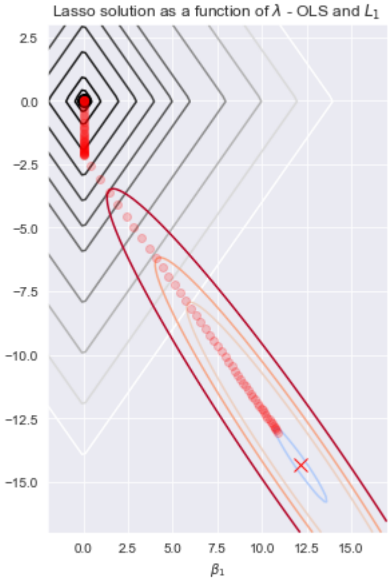
- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \frac{\|\theta\|_2^2}{2}$
- Also known as Tikhonov regularization or ridge regularization
- Gradient descent rule with a fixed learning rate becomes $\theta \leftarrow (1-\eta\kappa)\theta - \eta\nabla_\theta\mathcal{E}(S,\theta)$
- Tends to output networks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



Notes

L^1 regularization

- $\mathcal{P}(\theta) \stackrel{\text{def}}{=} \|\theta\|_1$
- Also called **Lasso regularization**
- Gradient descent rule becomes
  $\theta \leftarrow \theta - \eta\kappa \cdot \mathrm{sign}(\theta) - \eta\nabla_\theta\mathcal{E}(\mathcal{S}, \theta)$
- 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



Notes

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

Notes

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)

Notes

On weight decay

- Another regularization technique designed to compute networks with small weights
- Update rule: $\theta \leftarrow (1-\lambda)\theta - \eta\nabla_\theta\mathcal{E}(S,\theta)$
- Adam appears to perform much better with weight decay than with L^2-regularization (Loshchilov & Hutter - Decoupled weight decay regularization. 2019)

Notes

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

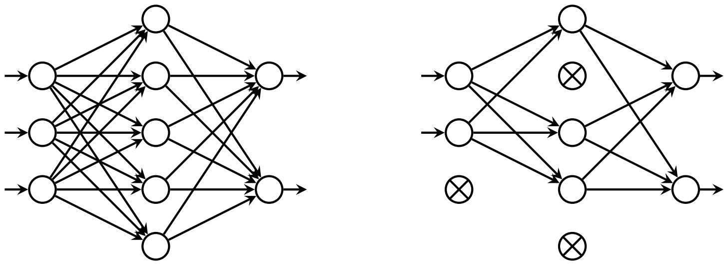
Input: n, number of steps between validation error evaluations
Input: p, number of observations of worsening validation errors before stop
Input: θ_0, initial parameters

$\theta \leftarrow \theta_0$, $\theta^* \leftarrow \theta$;
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^*

Notes

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?

Notes

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

Notes

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**:

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

Notes

On forward and backpropagation with dropout

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

Notes

Notes