Neural Networks II

Wei Li

Syracuse University

Spring 2024

OVERVIEW

Some Issues in Training Deep Neural Networks

Regularization

Approximation and Statistical Properties

Some Issues in Training Deep Neural Networks

Some Issues in Training Deep Neural Networks

The objective function $\mathcal{L}(\theta)$ is nonconvex, possessing many local minima; with deep networks, vanishing/exploding gradients are also issues

- the choice of activation functions (seen before)
- the choice of learning rates (and optimization algorithm)
- standardization of the inputs (and activations)
- the initialization of the weights

The complexity of the model needs special attention to avoid overfitting,

- early stopping
- regularization via penalty
- dropout
- residual connection
- data augmentation
- the architecture of the network (outside the scope)

Learning Rates

$$\mathcal{L}_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}, \theta\right), y^{(i)}\right)$$

How to set ϵ_t?

GD update: at the t-th iteration

$$\theta^{<t+1>} = \theta^{<t>} - \epsilon_t \frac{1}{B}\sum_{i\in\mathcal{B}} \nabla_\theta \mathcal{L}\left(f\left(x^{(i)}, \theta^{<t>}\right), y^{(i)}\right)$$

- $\mathcal{B}$ is a subset or a batch (taken at random without replacement from training data) of cardinality B (**batch size**).

Learning rates

A sufficient condition for SGD to converge is the Robbins-Monro Condition: let t denote iteration number or epoch number.

$$\frac{\sum_{t=1}^{\infty} \epsilon_t^2}{\sum_{t=1}^{\infty} \epsilon_t} \to 0 \quad \text{as } \epsilon_t \to 0$$

- Step decay: reduce the learning rate by some factor every few epochs.
- Exponential decay: $\epsilon_t = \epsilon_0 e^{-kt}$ where ϵ_0 and k are hyperparameters.
- Polynomial decay: $\epsilon_t = \frac{\epsilon_0}{(1+kt)^a}$ where , ϵ_0 and a, k are the hyperparameters.

- learning rate warmup: quickly starts with increasing learning rate, then decreases it slowly
- cyclical learning rate
- iterate averaging

$$\theta^{<t+1>} = \frac{1}{t}\sum_{i=1}^{t}\theta^{<i>} = \frac{1}{t}\theta^{<t>} + \frac{t-1}{t}\bar{\theta}^{<t>}$$

Improved optimization: Momentum

Plain-vanilla:

$$\Delta\theta^{<t>} \leftarrow \nabla_\theta \mathcal{L}(\theta^{<t>})$$
$$\theta^{<t+1>} \leftarrow \theta^{<t>} - \Delta\theta^{<t>}$$

Basic idea of momentum: not only move in the (opposite) direction of the gradient, but also move in the "averaged" direction of the last few updates. It helps overcoming the slow gradients in flat regions and escaping local minima.

Momentum:

$$v^{<t>} \leftarrow \gamma v^{<t-1>} + \nabla_\theta \mathcal{L}\left(\theta^{<t>}\right), \quad \gamma \in (0, 1)$$
$$\theta^{<t+1>} \leftarrow \theta^{<t>} - v^{<t>}$$

Momentum update

$$v^{<t>} \leftarrow \gamma v^{<t-1>} + \nabla_\theta \mathcal{L}\left(\theta^{<t>}\right), \quad \gamma \in (0,1)$$
$$\theta^{<t+1>} \leftarrow \theta^{<t>} - v^{<t>}$$

With momentum update, the parameter vector will build up velocity in any direction that has consistent gradient.

γ: exponential decay rate

The velocity effectively is exponentially decaying moving average of gradient

$$v^{<t>} = \sum_{j=1}^{t} \gamma^{t-j} \nabla_\theta \mathcal{L}\left(\theta^{<j>}\right)$$

Note: A look-ahead version of the algorithm is called "Nesterov Momentum", helping dampening oscillations.

Improved optimization: per-parameter adaptive learning

The following algorithms are improvements of Gradient Descent with Momentum, and additionally allow to adjust learning rates adaptively for different parameters.

- Adagrad (Adaptive Gradient Method)
- RMSProp (Root Mean Square Propagation)
- Adam (Adaptive Moment Estimation).

As of now, Adam (and its variant) is one of the most popular and successful algorithms in deep learning.

For more details:
https://weili-code.github.io/StatisticalSimulation/7_modern_optimization.html

Features normalization

input normalization: The whole training input are often normalized before they feed into the network.

- normalization of the input eliminates their unit effects.
- normalization of the input ensures the regularization affects the weights roughly uniformly. It also allows more systematic use of weights initialization.

batch normalization: the normalization is restrained to each mini-batch for each layer separately in the training process.

- activations within a layer when averaged over the samples in the minibatch has mean zero and unit variance

Features normalization (batch normalization)

At a given layer, for the i-th observation, we replace its activation vector $a^{(i)}$ with $\tilde{a}^{(i)}$:

$$\tilde{a}^{(i)} = \gamma \otimes \hat{a}^{(i)} + \beta$$

$$\hat{a}^{(i)} = \frac{a^{(i)} - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

$$\mu_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} a^{(i)}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left(a^{(i)} - \mu_{\mathcal{B}} \right)^2$$

- $\mathcal{B}$ is the minibatch containing example i
- $\mu_{\mathcal{B}}$, $\sigma_{\mathcal{B}}^2$ is the mean and variance of the activations for this batch
- $\hat{a}^{(i)}$ is the standardized activation vector
- $\tilde{a}^{(i)}$ is the shifted and scaled version (the output of the BN layer)
- β and γ are vectors of parameters for the BN layer.

The batch normalization is applied after linear layer, but either before activation or after activation (as above). During testing, each layer uses the mean and variance estimated from the entire training dataset to compute $\hat{a}^{(i)}$.

Weight initialization

Weight initialization has the following impact

- speed of convergence
- small values cause diminishing signals
- large values cause exploding signals

Three most common weight initialization:

- Initialize weights to zero
- Initialize weights to $\mathcal{N}(\mu = 0, \sigma^2)$
- Initialized weights to uniform$(-u, u)$

However, deterministic weight initialization should be avoided. Symmetry issues occur when multiple neurons in the same layer learn identical or highly similar features, preventing effective learning.

Random initialization can break the unit symmetry and is often used.

LeCun/He/Xavier/Glorot and Bengio Initialization: The underlying idea is to preserve the variance of activation values and gradients between layers; avoiding saturating or exploding values.

LeCun initialization:

$$\sigma\left(w_{i,j}^{[\ell]}\right) = \sqrt{\frac{1}{n^{[\ell-1]}}}$$

He initialization:

$$\sigma\left(w_{i,j}^{[\ell]}\right) = \sqrt{\frac{2}{n^{[\ell-1]}}}$$

Xavier initialization:

$$\sigma(w_{i,j}^{[\ell]}) = \sqrt{\frac{2}{n^{[\ell-1]} + n^{[\ell]}}}.$$

Glorot and Bengio initialization:

$$w_{i,j}^{[\ell]} \sim \mathrm{U}\left(-\frac{6}{n^{[\ell-1]} + n^{[\ell]}}, \frac{6}{n^{[\ell-1]} + n^{[\ell]}}\right)$$

Regularization

Regularization

As for any machine learning methods, overfitting may occur if a model becomes too complex. To regularize the deep neural network, the following strategies are often used:

- early stopping
- regularization via penalty
- dropout
- residual connection
- data augmentation
- the architecture of the network (outside the scope)

Early stopping

Early stopping: stop the training process when the error on the validation set starts to increase. This is heuristic approach.

- Split data: divide the data into at least two sets: a training set and a validation set.
- Train model: begin training the model using the training set, and periodically evaluate it on the validation set.
- Monitor performance: at each evaluation step, check the model's performance on the validation set.
- Decision to stop: if the performance on the validation set begins to worsen, or fails to improve for a specified number of training epochs (a patience parameter), this indicates that the model may be starting to overfit the training data.

ℓ_2-reguralization

regularization via penalty:

One of the most common method is ℓ_2-regularization.

Penalized empirical risk:

$$\tilde{\mathcal{L}}_n(\theta) := \mathcal{L}_n(\theta) + \lambda\Omega(\theta) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}, \theta\right), y^{(i)}\right) + \lambda\Omega(\theta)$$

where

$$\Omega(\theta) = \sum_{\ell}\sum_{i}\sum_{j}\left(w_{i,j}^{[\ell]}\right)^2 = \sum_{\ell}\left\|W^{[\ell]}\right\|_F^2$$

$$\tilde{\mathcal{L}}_n(\theta) := \mathcal{L}_n\left(W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}\right) + \frac{\lambda}{2}\sum_{\ell=1}^{L}\left\|W^{[\ell]}\right\|_F^2$$

- Without regularization: First get $\nabla_{W^{[\ell]}}\mathcal{L}_n$ from back propagation, then

$$W^{[\ell]} \leftarrow W^{[\ell]} - \epsilon\nabla_{W^{[\ell]}}\mathcal{L}_n$$

- With regularization:

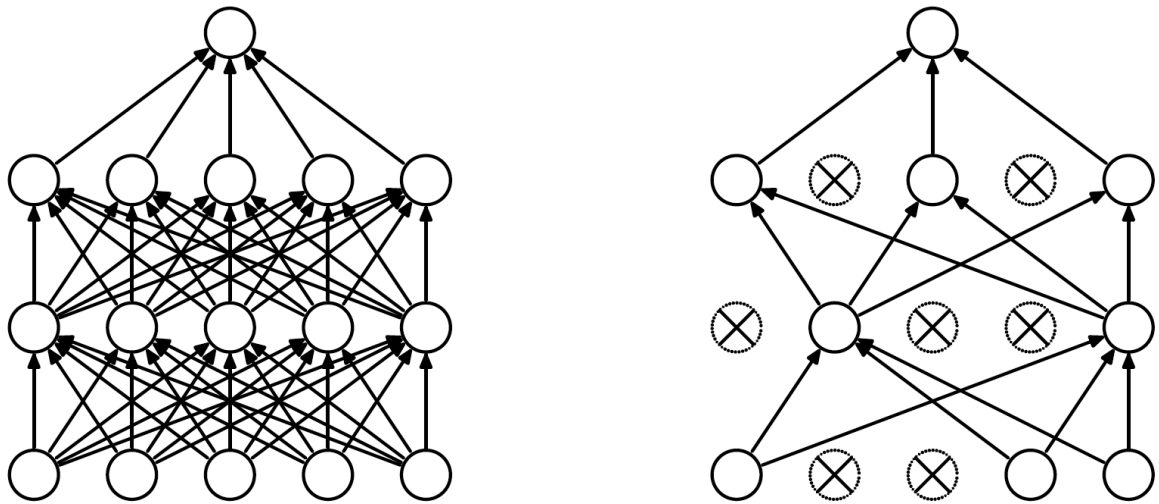
$$\nabla_{W^{[\ell]}}\tilde{\mathcal{L}}_n = \nabla_{W^{[\ell]}}\mathcal{L}_n + \lambda W^{[\ell]}$$
$$W^{[\ell]} \leftarrow W^{[\ell]} - \epsilon\nabla_{W^{[\ell]}}\tilde{\mathcal{L}}_n$$

which effects *weight decaying*

$$W^{[\ell]} \leftarrow (1-\epsilon\lambda)W^{[\ell]} - \epsilon\nabla_{W^{[\ell]}}\mathcal{L}_n$$

Dropout

Dropout is implemented in *a layer* such that, on an *individual example basis*, a certain proportion of its neurons are randomly turned off, along with their connections.



Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting.

- Dropout is not applied to the output layer.
- Dropout is applied in both forward and backpropagation during training stage.
- Dropout is not used during test stage (i.e. *inference* stage).

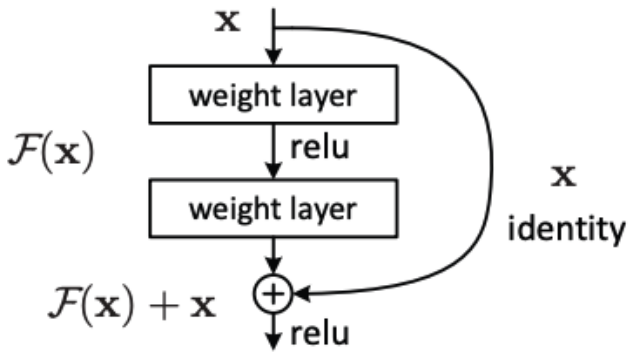
Suppose $100p\%$ nodes are randomly dropped in a layer, two options to implement

(option 1): During training, drop nodes; during testing (inference), do not drop nodes but multiply the activation values by $(1-p)$.

(option 2): During training, drop nodes *and* multiply the resulting activation values by $1/(1-p)$; without need to adjust during testing stage.

Residual connection (skip connection)

Skip connections help by creating an alternative shortcut path for the gradient during backpropagation. This direct path facilitates the flow of gradients through the network.



- $H(x)$ is the desired underlying original mapping by a few stacked layers
  - linear-activation-linear-$\cdots$-linear-activation-linear
- The residual learning approach reformulates the mapping learned as $F(x) := H(x) - x$
  - $F(x)$ is the residual mapping

He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition.

Data augmentation

Data augmentation involves artificially expanding the training dataset by creating modified versions of data points, which helps to prevent the model from overfitting and improves its ability to perform well on unseen data. Most common for image data, text data and audio data.

For example, common augmentation techniques for image data include:

- Rotation: rotating the images by different angles.
- Translation: shifting the images horizontally or vertically.
- Rescaling: adjusting the size of the images.
- Flipping: mirroring the images horizontally or vertically.
- Cropping: taking different cropped portions from the original image.
- Color alteration: adjusting brightness, contrast, saturation, or hue.

Approximation and Statistical Properties

Approximation properties

(Hornik et al.,1989; 1990; Leshno et al. 1993)

The **Universal Approximation Theorem**:

A feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (e.g., sigmoid, Relu) can approximate any suitably smooth function arbitrarily well, provided that the network is given enough hidden units.

Wider or deeper?

(Montufar et al 2014)

A feedforward network with a single layer is sufficient to represent any function, but the layer size may be infeasibly large. Shallow networks may fail to learn and generalize correctly if "too shallow".

In the asymptotic limit of many hidden layers, deep networks are able to separate their input space into exponentially more linear response regions than their shallow counterparts, with the same number of units.

Empirically, deeper networks seem to result in better generalization.

Convergence rates

(Schmidt-Hieber 2020)

$$Y_i = f(X_i) + \varepsilon_i, \quad X_i \in \mathbb{R}^p$$

Assumptions:

a general composition form on the true regression function f^*.

$$f^* = g_q \circ g_{q-1} \circ \cdots \circ g_1 \circ g_0$$

- $g_i : [a_i, b_i]^{d_i} \to [a_{i+1}, b_{i+1}]^{d_{i+1}}$.
- each g_i is a t_i-variate, α_i-smooth function.
  - t_i is not growing as levels deepens (required by minimax)
  - $t_i \ll d_i$

Using a **sparse** multi-layer feedforward network for f with

- relu activation
- bounded networks parameters values
- number of parameters exceed the sample size
  - with suitably tuning sparsity level s (see below)

The empirical minimizer using such a network f can achieves test MSE minimax rates ϕ_n (up to log factor)

$$\alpha_j^* := \alpha_j \prod_{\ell=j+1}^{q} (\alpha_\ell \wedge 1)$$

$$\phi_n := \max_{j=0,\ldots,q} n^{-\frac{2\alpha_j^*}{2\alpha_j^*+t_j}}$$

with depth scales with the sample size $\log n$; for the sparsity level

$$s \asymp n\phi_n \log(n).$$

special cases (with sparsity)

If the true function is of the generalized additive form:

$$f^*\left(X_1, \ldots, X_p\right) = \sum_{j=1}^{p} f_j\left(X_j\right)$$

or tensor-product form:

$$f^*(X) = \sum_{\ell=1}^{M} b_\ell \prod_{j=1}^{p} f_{j\ell}\left(X_j\right),$$

assuming each component function is α-smooth, the sparse deep NN with the sparsity level $s \asymp n^{1/(2\alpha+1)} \log(n)$ achieves the rate $n^{-\frac{2\alpha}{2\alpha+1}}$ up to log factor.

- compared with the usual minimax rate $n^{-2\alpha/(2\alpha+p)}$ (w/o sparsity)
- p does not show up (better than classical additive nonparametric regression estimator)

Under the given assumptions, the **sparse** deep NN circumvents the curse of dimensionality!