Deep Neural Networks

Dit-Yan Yeung

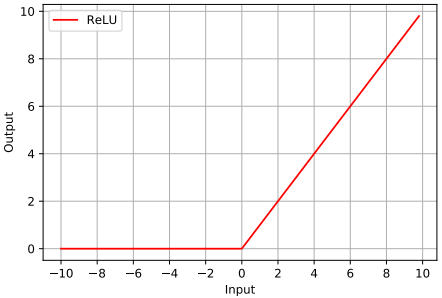
Department of Computer Science and Engineering
Hong Kong University of Science and Technology

COMP 4211: Machine Learning (Fall 2022)

Challenges of Training Deep Neural Networks

- The vanishing gradient problem makes the lower layers (i.e., the hidden layers closer to the input) of a deep neural network very difficult to train, because the gradients often get smaller and smaller in magnitude as the BP algorithm progresses down to the lower layers.
- The many network weights in a deep neural network make it easy to overfit the training data.
- It would be extremely time consuming to train a large network.

Rectified Linear Unit



- The rectifier is an activation function defined as:

$$g(x) = \max(0, x).$$

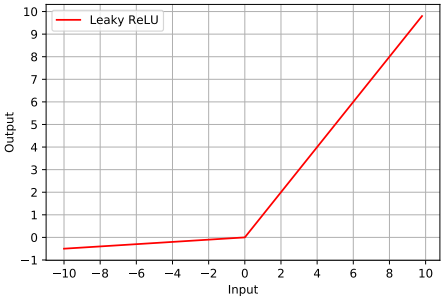
- A processing unit that uses the rectifier as activation function is called a rectified linear unit (ReLU).

Rectified Linear Unit (2)

- ReLU alleviates the vanishing gradient problem because it does not saturate for positive input values.
- Another advantage of ReLU is that it is quite fast to compute.
- However, the vanishing gradient problem is not completely addressed because the gradient is 0 for negative input values. When this happens, the output is 0 and the gradient is 0, making the dying ReLU unlikely to come back to life again.
- Also, although the function is continuous, it is not differentiable when the input is 0. Nevertheless, this usually does not cause any problem in practice.
- A smooth approximation to the rectifier is the softplus function:

$$g(x) = \log(1 + e^x).$$

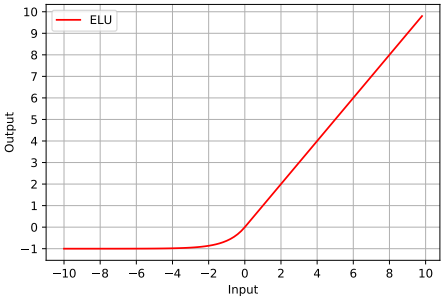
Leaky ReLU



- Leaky ReLU improves ReLU by allowing a small positive gradient for negative input values (e.g., $\alpha = 0.01$):

$$g(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise.} \end{cases}$$

Exponential Linear Unit



- The exponential linear unit (ELU) has the following activation function:

$$g(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{otherwise,} \end{cases}$$

where $\alpha > 0$ is a hyperparameter (e.g., initialized to 1).

Exponential Linear Unit (2)

- An advantage of ELU is that its mean activation value is closer to 0, which can be proved to enable faster learning.
- Unlike ReLU and its variants, the activation function of ELU is smooth everywhere, including around $x = 0$, which speeds up gradient descent because it does not bounce as much left and right of $x = 0$.
- The main drawback of the ELU activation function is that it is slower to compute than ReLU and its variants due to its use of the exponential function. This is not too much of a problem during training because the slower computation is compensated by the faster convergence, but during testing it will be considerably slower than ReLU.

Which Activation Function to Choose?

- If runtime performance is not a concern, in general the relative performance of different choices is:

$$\text{ELU} > \text{leaky ReLU (and its variants)} > \text{ReLU} > \tanh > \text{sigmoid}.$$

- If runtime performance is an important factor to consider, leaky ReLU will be a better choice than ELU.
- The hyperparameters in leaky ReLU and ELU may be learned to further boost the performance if the training set is sufficiently large (or else overfitting may occur).

Xavier Initialization

- Observations:
  - If the weights in a network start too small, then the signal shrinks as it passes through each layer until it is too tiny to be useful.
  - If the weights in a network start too large, then the signal grows as it passes through each layer until it is too massive to be useful.
- The weight initialization strategy called Xavier initialization (named after the author's first name) makes sure the weights are in a reasonable range of values so that the signals can reach deep into the network.
- The main idea is to let the distribution governing the initialization of weights depend on the number of input connections and the number of output connections for the layer whose weights are being initialized.

Xavier Initialization for Sigmoid Activation Function

- Let n_{in} and n_{out} denote the number of input connections and number of output connections, respectively, in a layer.
- The weights may be initialized randomly using a normal distribution or a uniform distribution.
- Zero-mean normal distribution with variance

$$\sigma^2 = \frac{2}{n_{in} + n_{out}}.$$

- Uniform distribution in $[-r, +r]$ where

$$r = \sqrt{\frac{6}{n_{in} + n_{out}}}.$$

He Initialization

- The He initialization (named after the author's last name), also called Kaiming initialization (named after the author's first name), is for ReLU and its variants, including ELU.
- Zero-mean normal distribution with variance

$$\sigma^2 = \frac{4}{n_{in} + n_{out}}.$$

- Uniform distribution in $[-r, +r]$ where

$$r = \sqrt{\frac{12}{n_{in} + n_{out}}}.$$

Internal Covariate Shift

- Although using a carefully designed weight initialization strategy along with a nonsaturating activation function such as leaky ReLU or ELU can significantly alleviate the vanishing/exploding gradient problem at the beginning of training, this problem may still arise later during training.
- Batch normalization is a technique for addressing the vanishing/exploding gradient problem by addressing the more general internal covariate shift problem (i.e., the distribution of the inputs of each layer changes during training as the network weights of the previous layers change).

Batch Normalization

- Before applying the activation function of each layer, the mean and variance of the inputs over the current mini-batch are evaluated to zero-center and normalize the inputs (hence the name 'batch normalization').
- Two new parameters, one for scaling and the other for shifting, are learned for each layer to restore its representation power.

Transform with Learnable Parameters

- Batch normalizing transform applied over a mini-batch:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

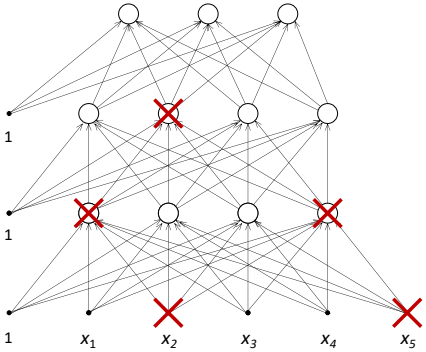
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- With batch normalization, the vanishing gradient problem can be significantly reduced even when saturating activation functions such as the sigmoid and hyperbolic tangent functions are used, without relying on carefully designed weight initialization and regularization.

Dropout

- Dropout is a regularization technique to prevent overfitting.
- At each training step, each hidden unit or input has a probability p of being temporarily dropped out (meaning that it will be entirely ignored during this training step, but it may become active again during the next step).
- The hyperparameter p, called the dropout rate, is typically set to 0.5 (but other values are also possible, even with different values for different layers). Sometimes it is more convenient to refer to the keep probability which is equal to $1 - p$.
- In general, a higher dropout rate is used if more serious overfitting is observed and for layers with more units (and hence more parameters to estimate).
- Dropout is only applied during training but not during testing.

Dropout at a Training Step



Why Does Dropout Work?

- Since hidden units and inputs are dropped out randomly, the network needs to learn to be more robust (i.e., less sensitive to slight changes in the inputs) by relying on more incoming units or inputs rather than just a few, effectively spreading out the weights to more units.
- An alternative way is to view dropout as ensemble learning (to be studied later), in that each training step "generates" one of 2^N possible networks, where N is the total number of droppable units or inputs, and trains the network on one example from the training set. The resulting neural network is essentially an ensemble of a large number of smaller networks.

An Important Technical Detail

- Suppose $p = 0.5$. Since dropout is not applied during testing, a hidden unit will on average be connected to twice (i.e., $1/(1-p)$ times) as many inputs as it was during training.
- Two alternative methods:
  - One way is to multiply the input weights of each unit by the keep probability $1-p$ after training.
  - Another way, called inverted dropout, is to divide the output of each unit by the keep probability during training.
- Strictly speaking these two methods are not equivalent but they both work well. Specifically, nothing needs to be done during testing and hence inference will not be slowed down.

Data Augmentation

- Overfitting can be reduced by having more training data, but it is not always possible to get more labeled training examples.
- Data augmentation is a regularization technique that generates new, realistic training instances from existing ones to increase the size of the training set effectively.
- Instead of wasting storage space and network bandwidth, it is preferable to generate new training instances on the fly during training.

Data Augmentation for Image Data

- Some commonly used operations:
  - Translation, rotation, or scaling by various amounts
  - Lighting with various contrasts
  - Cropping
  - Flipping horizontally.
- Some deep learning frameworks such as TensorFlow provide image manipulation operations for generating new training instances efficiently on the fly.

AdaGrad

- Stochastic gradient descent (SGD):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} L,$$

  where ∇ denotes the vector differential operator.
- AdaGrad improves SGD using an adaptive learning rate which decays faster for steeper dimensions.
- AdaGrad:

$$\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\mathbf{w}} L \circ \nabla_{\mathbf{w}} L$$
$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} L \oslash \sqrt{\mathbf{s} + \epsilon},$$

  where $\circ$ denotes the Hadamard product (a.k.a. elementwise multiplication), $\oslash$ denotes the Hadamard division (a.k.a. elementwise division), and ϵ is a smoothing term with a small value (e.g., 10^{-10}) for all dimensions to avoid division by 0.

AdaGrad (2)

- The first step of AdaGrad accumulates the square of the gradients into the vector $\mathbf{s}$: a dimension of $\mathbf{s}$ will become larger and larger if the loss function is steep along that dimension.
- The second step is similar to SGD except that there is a scaling factor $\sqrt{\mathbf{s} + \boldsymbol{\epsilon}}$: a dimension will decay its learning rate faster if the loss function is steep along that dimension.
- With the adaptive learning rate (without requiring manual tuning), AdaGrad helps point the resulting updates more directly towards the global optimum.
- Although AdaGrad performs well for simple quadratic problems, it often stops too early before reaching the global optimum when used for training neural networks.

RMSProp

- To overcome the problem of AdaGrad which decays too fast, RMSProp does not accumulate all the gradients from the beginning but only those from the recent iterations by introducing exponential decay in the first step.
- RMSProp:

$$\mathbf{s} \leftarrow \beta\,\mathbf{s} + (1-\beta)\nabla_{\mathbf{w}} L \circ \nabla_{\mathbf{w}} L$$
$$\mathbf{w} \leftarrow \mathbf{w} - \eta\,\nabla_{\mathbf{w}} L \oslash \sqrt{\mathbf{s} + \boldsymbol{\epsilon}},$$

where the decay rate β is typically set to 0.9.

Adam

- Adam (adaptive moment estimation) combines the ideas of momentum and RMSProp:
  - Like momentum, Adam keeps track of an exponentially decaying average of the past gradients.
  - Like RMSProp, it keeps track of an exponentially decaying average of the past squared gradients.
- Adam:

$$\begin{aligned}
\Delta\mathbf{w} &\leftarrow \beta_1 \Delta\mathbf{w} - (1-\beta_1)\nabla_{\mathbf{w}} L \\
\mathbf{s} &\leftarrow \beta_2 \mathbf{s} + (1-\beta_2)\nabla_{\mathbf{w}} L \circ \nabla_{\mathbf{w}} L \\
\Delta\mathbf{w} &\leftarrow \frac{\Delta\mathbf{w}}{1-{\beta_1}^t} \\
\mathbf{s} &\leftarrow \frac{\mathbf{s}}{1-{\beta_2}^t} \\
\mathbf{w} &\leftarrow \mathbf{w} + \eta \Delta\mathbf{w} \oslash \sqrt{\mathbf{s}+\epsilon},
\end{aligned}$$

where t is the iteration number.

A Point of Caution

- Although Adam generally works well and is faster than other methods, a study showed that adaptive optimization methods including AdaGrad, RMSProp and Adam can give solutions that generalize poorly on some data sets.
- So there is no single method that is always the best.

To Learn More...

- Neural architecture search
- Model compression