Deep Learning

Kaige Yang
University College London
Kaige.yang.11@ucl.ac.uk

Contents

Report Draft.

Abstract

Follow the book **Deep Learning** and **Deep Learning with Pytorch**.

1 Deep Forward Network

The goal of feed-forward network is to approximate some function $f^*(x)$. A feed-forward network defines a mapping $y = f(x, \theta)$ and learns the value of the parameters θ that result in the best approximation. If each layer is linear, the network as a whole would remain a linear function. Therefore, we need non-linearity called an activation function.

Universal approximation theorem:
A feed-forward network with a linear output layer and at least one hidden layer with any 'squashing' activation function (such as logistic sigmoid activation function) can approximate any Borel measureable function from one finite-dimensional space to another with any desired nonzero amount of error., provided that the network is given enough hidden units.

The largest difference between linear model and neural network is the non-linearity that causes most interesting loss functions to be non-convex. This means that neural network are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or convex optimization algorithms with global convergence guarantees.

An important aspect of the design of neural network is the choice of **cost function**. In most cases, our neural network defines a distribution $p(y|x, \theta)$ and we simply use the principle of maximum likelihood. That means we use the cross-entropy between the training data and the model's predictions as the cost function.

The output of a neuron is

$$h(x) = \phi(x^T w + b) \tag{1}$$

The outputs of a fully connected layer is

$$h(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b}) \tag{2}$$

where $\mathbf{X}$ represents the matrix of input features, $\mathbf{W}$ contains all the connection weights except for the ones from bias. The bias vector b contains all the connection weights between the bias neuron and the artificial neuron. It has one bias term per artificial neuron. The function ϕ is called the activation function.

Activation functions Logistic function

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \tag{3}$$

this has a well-defined nonzero derivative everywhere, allowing gradient descent to make some progress at every step.

The hyperbolic tangent function

$$tanh(z) = 2\sigma(2z) - 1 \tag{4}$$

This function is S-shaped, continuous and differetiable but its output ranges from -1 to 1, which tends to make each layer's output more or less centered around 0 at the beginning of training. This often helps speed up convergence.

The rectified unit function

$$ReLU(z) = \max(0, z) \tag{5}$$

It is continuous but not differentiable at $z = 0$. The derivative is 0 for $z < 0$. The advantage is fast to compute. Most importantly, the fact that it does not have a maximum output value also helps reduce some issues during gradient descent.It guarantees the output always be positive.

- ReLU (Rectified Linear Unit)
- Absolute value rectification
- Leaky ReLU

- Parametric ReLU (PRelU)
- Maxout units
- Logistic Sigmoid
- Hyperbolic Tangent
- Radial Basis Funtion (RBF)
- SoftPlus
- Hard Tanh

Architecture Design The overall structure of network: how many units it should have a how these units should be connected to each other.

For regression, the loss function typically is mean-squared error (MSE). if there are a lot outlier in the data set, you may prefer to use the mean absolute error (MAE) or Huber loss the combination of both. Huber loss makes it less sensitive to outliers than MSE and it is often more precise and converges faster than the MAE.

For classification, logistic function is used for binary classification, while softmax function is used to multiclass. The loss function is cross-entropy.

2 Training Neural Network

The **back-propagation** algorithm often simple called backprop, allows the information from the cost to then flow backward through the network in order to compute the gradient. In short, it is simple gradient descent using an efficient technique for computing the gradient automatically. The back-propagation algorithm is able to compute the gradient of the network's error with regards to every single model parameter and performs a regular gradient descent step. The whole process is repeated until the network converges to the solution.

In summary, for each training instance the back-propagation algorithm first makes a prediction (forward pass) measures the error, then goes through each layer in reverse to measure the gradient of each weights and finally tweaks the connection weights to reduce the error (gradient descent).

[blue: it is important to initialize all weights randomly, or else training will fail. if all weights and bias are initialized as zero, the back-propagation will affect them in the same way, they will remain identical. If you randomly initialize the weights, you break the symmetry and allow back-propagation to train a diverse team of neurons.]

vanishing gradient and exploding gradient problem

Backpropagation works from the output layer to input layer, propagating the error gradient on the way. the parameters are updated by gradient descent.

vanishing gradient: gradients often get smaller and smaller as the algorithm progress down to the lower layers. thus, gradient descent leaves the parameters at lower layers virtually unchanged and training never converges to a good solution.

One of the reason of these issues is due to the choice of activation function. For example, looking at the logistic activation function when inputs becomes large, the function saturates at 0 or 1, with a derivative extremely close to 0. Thus, when backpropagation gradient, there is really nothing left for the lower layers.

ReLU behaves much better in deep learning mostly due to it does not saturate for positive values. However, it suffers dying ReLU problem.

Dying ReLU: during training, some neurons effectively die, meaning they stop outputting anything other than 0. This happens when a neuron's weights is updated such that the weighted sum of its input is negative, it will starts to output 0. When this happen, the neuron is unlikely to come back to live since the gradient of the ReLU is 0 when the input is negative.

One way to solve is leaky ReLU

$$LeakyReLU(\alpha, z) = \max(\alpha z, z) \tag{6}$$

α is the slope of the function for $z < 0$. This small slope ensure that leaky ReLU never die.

Another active function is ELU

$$ELU(\alpha, z) = \alpha(\exp(z) - 1),\ if\ z < 0;\ \ z\ if\ z \geq 0 \tag{7}$$

The preferred ranking of activation function

$$ELU > LeakyReLU > ReLU > Tanh > logistic \tag{8}$$

exploding gradient: the gradient can grow bigger and bigger, may layers get insanely large weight updates and the algorithm diverges, which is mostly encountered in recurrent neural network.

Solutions to solve these issues
One author argue that we need the variance of the outputs of each layer to be equal to the variance of its input and we also need the gradients to have equal variance before and after flowing through a layer in the reverse direction.

This is actually not possible to guarantee both unless the layer has an equal number of input and output connections.

They propose Xavier initialization for logistic activation function

$$\mathcal{N}(0, \frac{2}{\sqrt{n_{input} + n_{output}}} \tag{9}$$

or

$$\mathcal{U}(-\sqrt{\frac{6}{n_{input} + n_{output}}}, \sqrt{\frac{6}{n_{input} + n_{output}}}) \tag{10}$$

This initialization strategy speed up training considerly.

The He initialization strategy for ReLU and ELU

$$\mathcal{N}(0, \sqrt{2}\frac{2}{\sqrt{n_{input} + n_{output}}} \tag{11}$$

or

$$\mathcal{U}(-\sqrt{2}\sqrt{\frac{6}{n_{input} + n_{output}}}, \sqrt{2}\sqrt{\frac{6}{n_{input} + n_{output}}}) \tag{12}$$

Although He initialization and ELU activation function reduce the vanishing/exploding gradients problems at the beginning of training, it does not guarantee that they will not come back.

Batch Normalization
Batch normalization is proposed to address the vanishing / exploding gradient problems and more internal covariate shift problem (ICS).

ICS: the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.

Batch normalization: adding an operation in the model before the activation function of each layer, simply zero-centering and normalizing the inputs. Then scaling and shifting the results using two new parameters (γ, β) which are learned by the model. The mean and variance is estimated based on the current mini-batch.

$$\mu = \frac{1}{b}\sum_{i=1}^{m} x_i \tag{13}$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2 \tag{14}$$

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{15}$$

$$x_i = \gamma\hat{x}_i + \beta \tag{16}$$

At test time, there is no mini-batch, so instead you simply use the whole training set's mean and standard deviation.

Batch normalization also acts a regularizer, reducing the need for other regularization.

cons:
Due to the need of learning γ and β, the training is slow at first. the network makes slower prediction due to the extra computation required at each layer.

The vanishing gradient problem is strongly reduced. the network is less sensitive to initialization. The learning rate is speed up.

Gradient clipping
A popular technique to lessen the exploding gradients problem is to simple clip the gradient during training so that they never exceed some threshold mostly useful for recurrent neural network. This is called gradient clipping. Batch normalization can also reduce the issue.

pre-trained layers
It not only speed up the training considerably, but also requires much less training data.

Upsupervised pretraining
pretraining on an auxiliary task

Learning schedules

- piece-wise constant learning rate
- performance schedule
- exponential scheduling
- power scheduling

3 Regularization

DN N typically have many parameters, thus it has incredible amount of freedom and can fit a huge variety of complex datasets. This will lead to overfitting the data. **Regularization techniques**:
early stopping: just interrupt the training when its performance on the validation set starts dropping.

$l1$ **and** $l2$ **regularization**
Dropout: at every training step, all neurons except output neurons have a probability p of being ignored during this training step, but may be active during the next step. After training, neurons do not get dropped anymore.

Neurons trained with dropout cannot co-adapt with their neighboring neurons; they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons; they must pay attention to each of their input neurons. They end up being less sensitive to slight changes in the inputs. In the end you get a more robust network that generalizes better.

Max-norm regularization: it constrains the weights w of the incoming connections such that $||w||_2 \leq r$. Reducing r increasing the amount of regularization and helps reduce overfitting. Max-norm regularization can also help alleviate the vanishing/exploding gradients problems.

Data Augmentation: this technique generates new realistic data to reduce overfitting. Simple adding noise will not work, the modifications you apply should be learned.

4 Optimization for Training Deep Models

Optimization: Find the parameters θ of a neural network that significantly reduce a cost function, which typically included a performance measure evaluated on the entire training set as well as additional regularization terms.

How learning differs from Pure Optimization
Challenges in neural Network Optimization

Algorithms

- Stochastic Gradient Descent (SGD)
- Momentum
- Nesterov Momentum
- AdaGrad
- RMSProp
- Adam
- Newton's Method
- Conjugate Gradients
- BFGS
- Limited Memory BFGS

Parameter initialization
Other Techniques

- Batch Normalization
- Coordinate Descent
- Polyak Averaging
- Supervised Pretraining
- Continuation Methods and Curriculum Learning
- Design models to ease optimization.

5 Convolutions Network

Convolutional layer: neurons in the first layer are not connected to every pixel in the input image, but only to pixels in their receptive layer. In turn, each neurons in the second layer is connected only to neurons located within a small rectangle in the first layer.

This architecture allows the network to concentrate on low-level features in the first hidden layer, then assemble them into higher-level features in the next hidden layers and so on.

zero padding: to make sure each neurons has the same size of receptive field.
stride: the distance between two consecutive receptive fields is called the stride.

The output of a neuron in a convolutional layer

$$z_{i,j,k} = b_k + \sum_{u=1}^{f_h} \sum_{v=1}^{f_w} \sum_{k'=1}^{f_{n'}} x_{i',j',k'}, w_{u,v,k',k} \tag{17}$$

Pooling layer: Each neurons in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. However, the pooling neurons has no weights, all it does is aggregate the input using an aggregation function such as max or mean.

The goal is to sub-sample the input image in order to reduce the computational load and the number of parameters. This reduces the risk of over-fitting.

CNN architectures: Typical CNN stack a few convolutional layers then a pooling layer, then another few convolutional layers then another pooling layer. At the top of stack, a regular feed-forward neural network is added, composed of a few fully connected layers and final layer outputs the prediction soft-max layer that output the estimated class probabilities.

- AlexNet
- GoogleNet

- ResNet

The Convolution Operation

$$s(t) = \int x(a)w(t-a)da = (x * w)(a) \tag{18}$$

x is the input and w is the kernel.

The discrete convolution is defined as

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{19}$$

In machine learning applications, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. For example, we have input image I and a two dimensional kernel K

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{20}$$

Convolution is commutative

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{21}$$

Many neural network libraries implement a related function called cross-correlation, which is the same as convolution

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{22}$$

6 Recurrent and Recursive Net

Recurrent neurons: The recurrent neuron receives the input as well as its own output from the previous time steps. Each neuron has two set of weights one for input one for output from previous time steps.

$$y_t = \phi(x_t^T w_1 + y_{t-1}^T w_2 + b) \tag{23}$$

Recurrent neuron layer: At each time step t, every neurons receives both the input vector and output vector from the previous time step.

$$Y_t = \phi(X_t^T w_1 + Y_{t-1}^T W_2 + b) \tag{24}$$

memory cell:
As the output of recurrent neuron at time step t is a function of the inputs from previous time steps. We can say it has a form of memory.

A part of neural network preserves some states across time steps is called a memory cell (cell). In general, a cell's state at time t denoted as h_t is a function of the input at the time step x_t and the state at previous time step $h_t = f(h_{t-1}, x_t)$. The output at time t is y_t. For simple recurrent neuron $y_t = h_t$. But, there exists more complex form of cell.

Forms of RNN

- Sequence to sequence
- Sequence to vector
- vector to sequence
- sequence to vector + vector to sequence (encoder-decoder)

train RNN:
To train an RNN, the trick is to unroll it through time adn then simple user backpropagation. This is called BPTT.

The difficulty of training over many time steps:
To train RNN on long sequence, it suffers from the vanishing/exploding gradient problem. Many tricks can be used: initialization, non-saturating activation function, batch normalization, gradient clipping and faster optimizer.

Another trick is truncated backpropagation through time. to unroll the RNN only over a limited number of time steps during training.

Another problem is the memory loss. Due to the transformation that the data goes through when traversing an RNN, some information is loss after each time step. For example, the current state contains no information of the first input.

To solve this problem, many cells are proposed.

LSTM: performs better, training will converge faster and it detects long-term dependencies in the data. LSTM maintains two state vectors: h_t as short-term state, c_t as long-term state.

The key idea is that the network can learn what to store in the long-term state, what to throw away, and what to read from it.

$$i_t = \sigma(W_{xi}^T x_t + W_{hi}^T h_{t-1} + b_i) \tag{25}$$

$$f_t = \sigma(W_{xf}^T x_t + W_{hf}^T h_{t-1} + b_f) \tag{26}$$

$$o_t = \sigma(W_{xo}^T x_t + W_{ho}^T h_{t-1} + b_o) \tag{27}$$

$$g_t = tanh(W_{xg}^T x_t + W_{hg}^T h_{t-1} + b_g) \tag{28}$$

$$c_t = f_t \otimes c_{t-1} + i_t \otimes g_t \tag{29}$$

$$y_t = h_t = o_t \otimes tanh(c_t) \tag{30}$$

GRU cell
The gated recurrent cell is a simplified version of LSTM.

- Two state vectors are. merged into a single vector h_t

$$z_t = \sigma(W_{xz}^T x_t + W_{hz}^T h_{t-1}) \tag{31}$$

$$r_t = \sigma(W_{xr}^T x_t + W_{hr}^T h_{t-1}) \tag{32}$$

$$g_t = tanh(W_{xg}^T x_t + W_{hg}^T (r_t \otimes h_{t-1})) \tag{33}$$

$$h_t = (1 - z_t) \otimes tanh(W_{xg}^T h_{t-1} + z_t \otimes g_t) \tag{34}$$

7 Autoencoder

Autoencoder is a neural network capable of learning representation of the input, which is useful form dimensionality reduction, being feature selector, generative model.

The loss function is reconstruction error. The number of output layer and input layer are the same.

Autoencoder has the shape of sandwich. **tying weights**: due to the symmetrical shape, the weight of the encoder can be shared with decoder. This halves the number of weights, speeding up the training and limiting the risk of overfitting.

Visualizing features Once trained the autoencoder, how to visulizing features.

- Consider each neurons in every hidden layer, and find the training instances that activate it the most.

- Create a image where a pixel's intensity corresponds to the weight of the connection to the given neuron.
- Feed the autoencoder a random input image, measure the activation of the neuron and then perform back propagation to tweak the image in such as way the the neuron will activate even more. The image will gradually turn into the most exciting image.

How to make sure autoencoder learn interesting features

- control the size of coding layer, making it undercomplete.
- Add noise to the input.
- Sparse autoencoder: adding an appropriate term to the cost function: KL divergence between the target sparsity and the actual sparsity

$$D_{KL}(p||q) = p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q} \tag{35}$$

To speed up training the reconstruction error measured b MSE can be replaced by cross-entropy. First normalize te input to [0,1], use logistic function at the output layer.

Variational autoencoders
Variational autoencoder shares the same form of autoencoder with one twist: instead of directly producing a coder for a given input, the encoder produces a mean coding μ and a standard deviation σ. The actual coding is then sampled randomly from a Gaussian distribution. After that the decoder just decoder the sampled coding directly.

The loss function is composed of two part: the reconstruction error and KL divergence between the learned Gaussian and target Gaussian.

Other Autoencoders

- Contractive autoencoder: two similar input are encoded similarly.
- Adversarial autoencoders: One network is trained to reproduce its input, and at the same time another is trained to find inputs at the first network is unable to properly reconstruct. This pushes the first autoencoder to learning robust coding.

8 DNN for Regression

9 DNN for Classification

10 DNN for Image Classification

11 DNN for Time-series Prediction

References