CS F425: Deep Learning

10 Backpropagation Training

Dr. Kamlesh Tiwari
Assistant Professor, Department of CSIS,
BITS Pilani, Pilani Campus, Rajasthan-333031 INDIA

Feb 10, 2023 ON-CAMPUS Campus @ BITS-Pilani [Jan-May 2023]

http://ktiwari.in/dl

Recap: Training Multilayer Networks with Backpropagation

- Single perceptron can only express linear decision surface
- We need units whose output is a nonlinear function of its inputs AND is also differentiable (using Neuron not ~~Perceptron~~)
  $o(\vec{x}) = \sigma(\vec{w}.\vec{x})$ where $\sigma(y) = \frac{1}{1+e^{-y}}$

Backpropagation algorithm learns weights
for a fixed set of units and interconnections

- It employs gradient descent to minimize the error between the network output values and the target values for these outputs
- Let Error function is redefined as

$$E(\vec{w}) = \frac{1}{2}\sum_{d\in D}\sum_{k\in outputs}(t_{kd} - o_{kd})^2$$

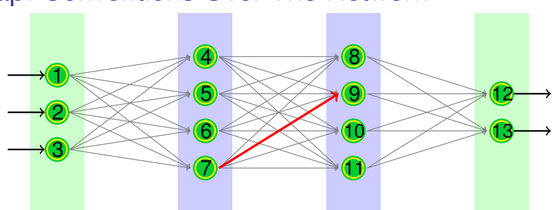
Recap: Backpropagation (for 2 layers)

Algorithm 1: Backpropagation(D, η, n_{in}, n_{out}, n_{hidden})

```
Create the feedforward network with n_in, n_out, n_hidden layers
Randomly initialize weights to small values ∈ [−0.05, +0.05]
repeat
    for each <x⃗,t⃗>∈D do
        o_u = get output from network ∀ unit u
        δ_k = o_k(1 − o_k)(t_k − o_k) for all output unit k
        δ_h = o_h(1 − o_h) Σ_{k∈outputs}(w_kh δ_k) for all hidden unit h
        w_ji = w_ji + △w_ji    where △w_ji = η δ_j x_ji
until converge;
```

- Recall error function is $E(\vec{w}) = \frac{1}{2}\sum_{d\in D}\sum_{k\in outputs}(t_{kd} - o_{kd})^2$
- For a single training example $E_d(\vec{w}) = \frac{1}{2}\sum_{k\in outputs}(t_k - o_k)^2$
- Weight w_{ji} is updated by adding $\triangle\, w_{ji} = -\eta\frac{\partial E_d}{\partial w_{ji}}$

Recap: Conventions Over The Network



x_{ji}	ith input to unit j (x_{94} is highlighted)
w_{ji}	weight associated with ith input to unit j
net_j	be $\sum_i w_{ji}x_{ji}$ the weighted sum of input for unit j
o_j	output computed by unit j. Consider it as $\sigma(net_j)$
t_j	target output for unit j
$outputs$	set of units in final layer ($\{12, 13\}$ in our case)
$Downstream(j)$	units whose immediate input is the output of unit j

We are interested in $\frac{\partial E_d}{\partial w_{ji}}$ it is $\frac{\partial E_d}{\partial net_j} \times \frac{\partial net_j}{\partial w_{ji}}$ and therefore, $\frac{\partial E_d}{\partial net_j} \times x_{ji}$

Value of $\frac{\partial E_d}{\partial net_j}$ for output units

- $\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \times \frac{\partial o_j}{\partial net_j}$
- $\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k\in outputs}(t_k - o_k)^2 = \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 = -(t_j - o_j)$
- Note that $o_j = \sigma(net_j)$ therefore $\frac{\partial o_j}{\partial net_j}$ is derivative of sigmoid

$$\begin{aligned}\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\frac{1}{1+e^{-x}} = (-1)(1+e^{-x})^{-2}\frac{d}{dx}(1+e^{-x})\\ &= (-1)(1+e^{-x})^{-2}(0-e^{-x})\\ &= \frac{1}{1+e^{-x}} \times \frac{e^{-x}+1-1}{1+e^{-x}} = \sigma(x)(1-\sigma(x))\end{aligned}$$

- As a result $\frac{\partial o_j}{\partial net_j} = \frac{\partial\sigma(net_j)}{\partial net_j} = \sigma(net_j)(1-\sigma(net_j)) = o_j(1-o_j)$
- $\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)o_j(1-o_j)$
- Term $(t_j - o_j)o_j(1-o_j)$ is treated as δ_j

Therefore, $\triangle\, w_{ji} = -\eta\frac{\partial E_d}{\partial net_j} \times x_{ji} = \boxed{\eta(t_j - o_j)o_j(1-o_j)x_{ji}}$

Value of $\frac{\partial E_d}{\partial net_j}$ for hidden units

$$\begin{aligned}\frac{\partial E_d}{\partial net_j} &= \sum_{k\in Downstream(j)}\frac{\partial E_d}{\partial net_k} \times \frac{\partial net_k}{\partial net_j}\\ &= \sum_{k\in Downstream(j)} -\delta_k \times \frac{\partial net_k}{\partial net_j}\\ &= \sum_{k\in Downstream(j)} -\delta_k \times \frac{\partial net_k}{\partial o_j} \times \frac{\partial o_j}{\partial net_j}\\ &= \sum_{k\in Downstream(j)} -\delta_k \times w_{kj} \times o_j(1-o_j)\end{aligned}$$

- δ_j being $-\frac{\partial E_d}{\partial net_j} = o_j(1-o_j)\sum_{k\in Downstream(j)}\delta_k \times w_{kj}$

Therefore, $\triangle\, w_{ji} = \eta\delta_j x_{ji} = \boxed{\eta(o_j(1-o_j)\sum_{k\in Downstream(j)}\delta_k \times w_{kj})x_{ji}}$

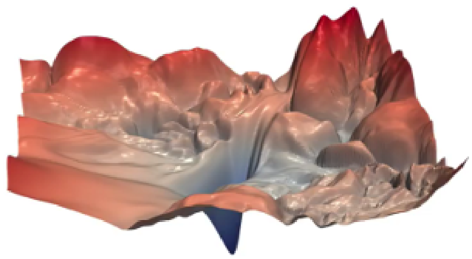
Note: $net_k = w_{k1}x_{k1} + w_{k2}x_{k2} + \ldots + w_{kj}x_{kj} + \ldots$; as o_j is input to kth unit so $o_j = x_{kj}$; so $\frac{\partial net_k}{\partial o_j} = w_{kj}$

The Backpropagation Algorithm

Algorithm 2: Backpropagation(D, η, n_{in}, n_{out}, n_{hidden})

Create the feedforward network with n_{in}, n_{out}, n_{hidden} layers
Randomly initialize weights to small values $\in [-0.05, +0.05]$
repeat
  for *each* $<\vec{x},\vec{t}>\in D$ **do**
    o_u = get output from network $\forall$ unit u
    $\delta_k = o_k(1-o_k)(t_k-o_k)$ for all output unit k
    $\delta_h = o_h(1-o_h)\sum_{k\in outputs}(w_{kh}\delta_k)$ for all hidden unit h
    $w_{ji} = w_{ji} + \triangle w_{ji}$ where $\triangle w_{ji} = \eta\delta_j x_{ji}$
until *converge*;

Loss Landscape [1]



Backpropagation over multilayer networks converge to a local minimum, NOT necessarily to the global minima

[1] Visualizing the Loss Landscape of Neural Nets, Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein, 2018

Backpropagation

- **Adding Momentum:** weight update during n^{th} iteration depend partially on the update that occurred during the $(n-1)^{th}$ iteration

$$\triangle\ w_{ji}(n) = \eta\delta_j x_{ji} + \alpha\ \triangle\ w_{ji}(n-1)$$

- **Learning in arbitrary acyclic network:** for feed-forward networks of arbitrary depth, δ_r value for a unit in hidden layer is determined as

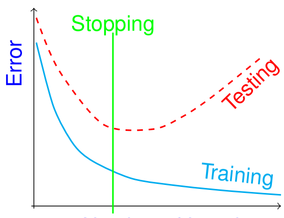
$$\delta_r = o_r(1-o_r)\sum_{s\in Downstream(r)} w_{sr}\times\delta_s$$

Backpropagation

- Result of Backpropagation over multilayer networks is only guaranteed to converge toward some local minimum and not necessarily to the global minimum error
- No methods are known to predict with certainty when local minima will cause difficulties
- Suggested to use momentum, true gradient descent or multiple networks (initialized with different random weights)
- Any boolean function can precisely be represented by some network having only **two** layers of units (Cybenko 1989; Hornik et al. 1989)
- Every bounded continuous function can be approximated with arbitrarily small error (under a finite norm) by a network with two layers of units
- Any function can be approximated to arbitrary accuracy by a network with three layers of units (Cybenko 1988)

Generalization, Overfitting, and Stopping Criterion

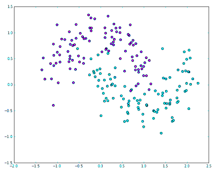
Continue training until the error on the training examples falls below some predetermined threshold could be a poor strategy

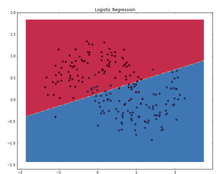


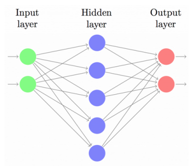
- Weight decay or use of validation set (k-fold ?) is suggested
- Input or output encoding can be used

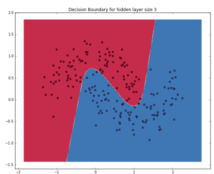
Coding Example [2]

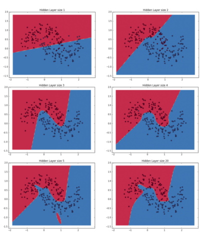
Consider two class data
Logistic Regression
Neural Network
Decision Boundary
Bigger hidden layer











[2] https://github.com/dennybritz/nn-from-scratch/blob/master/nn-from-scratch.ipynb

An Example Code [1/4] [3]

```
import matplotlib.pyplot as plt
import numpy as np
import sklearn
import sklearn.datasets
import sklearn.linear_model
import matplotlib

matplotlib.rcParams['figure.figsize'] = (10.0, 8.0)
np.random.seed(0)
X, y = sklearn.datasets.make_moons(200, noise=0.20)
plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=plt.cm.Spectral)

def plot_decision_boundary(pred_func):
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    h = 0.01
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h)
    Z = pred_func(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral)

plot_decision_boundary(lambda x: clf.predict(x))
plt.title("Logistic Regression")
plt.show()
```

[3] https://github.com/dennybritz/nn-from-scratch/blob/master/nn-from-scratch.ipynb

An Example Code [2/4] [4]

```
#### Implementing NN ####
num_examples = len(X) # training set size
nn_input_dim = 2 # input layer dimensionality
nn_output_dim = 2 # output layer dimensionality

epsilon = 0.01 # learning rate for gradient descent
reg_lambda = 0.01 # regularization strength

def calculate_loss(model):
    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']
    z1 = X.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    corect_logprobs = -np.log(probs[range(num_examples), y])
    data_loss = np.sum(corect_logprobs)
    data_loss += reg_lambda/2 * (np.sum(np.square(W1)) + np.sum(np.square(W2)))
    return 1./num_examples * data_loss

def predict(model, x):
    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']
    z1 = x.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    return np.argmax(probs, axis=1)
```

[4] https://github.com/dennybritz/nn-from-scratch/blob/master/nn-from-scratch.ipynb

An Example Code [3/4] [5]

```
def build_model(nn_hdim, num_passes=20000, print_loss=False):
    np.random.seed(0)
    W1 = np.random.randn(nn_input_dim, nn_hdim) / np.sqrt(nn_input_dim)
    b1 = np.zeros((1, nn_hdim))
    W2 = np.random.randn(nn_hdim, nn_output_dim) / np.sqrt(nn_hdim)
    b2 = np.zeros((1, nn_output_dim))
    model = {}
    for i in range(0, num_passes):
        z1 = X.dot(W1) + b1
        a1 = np.tanh(z1)
        z2 = a1.dot(W2) + b2
        exp_scores = np.exp(z2)
        probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
        delta3 = probs
        delta3[range(num_examples), y] -= 1
        dW2 = (a1.T).dot(delta3)
        db2 = np.sum(delta3, axis=0, keepdims=True)
        delta2 = delta3.dot(W2.T) * (1 - np.power(a1, 2))
        dW1 = np.dot(X.T, delta2)
        db1 = np.sum(delta2, axis=0)
        dW2 += reg_lambda * W2
        dW1 += reg_lambda * W1
        W1 += -epsilon * dW1
        b1 += -epsilon * db1
        W2 += -epsilon * dW2
        b2 += -epsilon * db2
        model = { 'W1': W1, 'b1': b1, 'W2': W2, 'b2': b2}
    return model
```

[5] https://github.com/dennybritz/nn-from-scratch/blob/master/nn-from-scratch.ipynb

An Example Code [4/4] [6]

```
# Build a model with a 3-dimensional hidden layer
model = build_model(3, print_loss=True)

# Plot the decision boundary
plot_decision_boundary(lambda x: predict(model, x))
plt.title("Decision Boundary for hidden layer size 3")

plt.show()

######### Varying the number of hidden layers ##########
plt.figure(figsize=(16, 32))
hidden_layer_dimensions = [1, 2, 3, 4, 5, 20, 50]
for i, nn_hdim in enumerate(hidden_layer_dimensions):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer size %d' % nn_hdim)
    model = build_model(nn_hdim)
    plot_decision_boundary(lambda x: predict(model, x))
plt.show()
```

[6] https://github.com/dennybritz/nn-from-scratch/blob/master/nn-from-scratch.ipynb

Thank You!

Thank you very much for your attention!