Introduction to Deep Learning

Machine Learning

Linear Regression

$$\hat{y}_i = \sum_{j=1}^{d} x_{ij}\theta_j$$

$d =$ input dimension
$x =$ input data
$\theta =$ weights
Loss function:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

in matrix notation:

$$J(\theta) = (\mathbf{X}\theta - \mathbf{y})^T(\mathbf{X}\theta - \mathbf{y})$$

differentiate:

$$\frac{\partial J(\theta)}{\partial \theta} = 2\mathbf{X}^T\mathbf{X}\theta - 2\mathbf{X}^T\mathbf{y} = 0$$

$$\theta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Analytical solution for a convex problem.

Logistic Regression

$$\hat{\mathbf{y}} = p(\mathbf{y} = 1|\mathbf{X}, \theta) = \prod_{i=1}^{n} p(y_i = 1|\mathbf{x}_i, \theta)$$

where

$$\hat{y}_i = \sigma(\mathbf{x}_i\theta)$$

σ is the sigmoid/logistic function.
Loss function (binary cross entropy loss):

$$L(\hat{y}_i, y_i) = -[y_i \, log \, \hat{y}_i + (1 - y_i) \, log(1 - \hat{y}_i]$$

No analytical solution. Iterative method needed (gradient descent).

Loss Functions

Measure the goodness of the prediction.
Large loss indicates bad performance.
Choice of loss function depends on problem.

Regression Loss

L1 Loss:

$$L(y, \hat{y}; \theta) = \frac{1}{n} \sum_i^n ||y_i - \hat{y}_i||_1$$

MSE Loss:

$$L(y, \hat{y}; \theta) = \frac{1}{n} \sum_i^n ||y_i - \hat{y}_i||_2^2$$

where
L1 norm (Manhattan distance):

$$||x||_1 = \sum |x_i|$$

- Promotes sparsity
- More robust to outliers

L2 norm (Euclidean distance):

$$||x||_2 = \sqrt{\sum x_i^2}$$

- Differentiable
- Penalizes large errors more heavily

Squared L2 norm:

$$||x||_2^2 = \sum x_i^2$$

- Differentiable

- Computationally efficient

Classification Loss

Binary Cross Entropy loss (binary classification):

$$L(y, \hat{y}; \theta) = -\frac{1}{n}\sum_{i}^{n}[y_i log\hat{y}_i + (1 - y_i)log(1 - \hat{y}_i)]$$

y is a probability output ($[0, 1]$) of the first class. Therefore, $1 - y$ is the probability of the second class. Works well with a sigmoind activation in the last layer.
Log penalizes bad predictions: $log\, 0 = -\infty$, $log\, 1 = 0$ (sort of inverts the prediction $\hat{y}$).
Encourages the model to make confident predictions close to 0 or 1.
Cross Entropy loss (multi-class classification):

$$L(y, \hat{y}; \theta) = -\sum_{i=1}^{n}\sum_{k=1}^{k}(y_{ik} \cdot log\, \hat{y}_{ik})$$

The second sum is a generalization of the binary cross entropy case for multiple classes.

Backpropagation

Backpropagation aims to minimize the loss function by adjusting network weights and biases. The level of adjustment is determined by the gradients of the loss function with respect to those parameters.
Neural network is a chain of functions: linear transformations and activation functions.
During backpropagation, the gradients of each linear transformation are calculated using the chain rule, starting from the derivative of the loss function and going backwards through the network.
Chain rule:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

or

$$h'(x) = f'(g(x))g'(x)$$

Calculate the gradients of the last layer:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial w_{ij}}$$

where z are the outputs of the layer before the activation function.
For any weight update in the hidden layers, sum all gradients of all neurons that are influenced by that weight.

Useful Derivatives

ReLU:

$$\frac{\partial ReLU}{\partial x} = if\ x < 0\ then\ 0;\ if\ x > 0\ then\ 1$$

Sigmoid:

$$\frac{\partial \sigma}{\partial x} = \sigma(x)(1 - \sigma(x))$$

Gradient descent

Gradient update step of the weight in direction of negative gradient (for a single training sample):

$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta L_i(\theta^k, x_i, y_i)$$

where α is the learning rate, typically very small.
Update step for n samples:

$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta L(\theta^k, x_{\{1..n\}}, y_{\{1..n\}})$$

where the gradient is the average over the residuals:

$$\nabla_\theta L(\theta^k, x_{\{1..n\}}, y_{\{1..n\}}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta L_i(\theta^k, x_i, y_i)$$

Potentially, the learning rate α could be calculated using Line Search:

$$arg\, min_\alpha L(\theta^k - \alpha \nabla_\theta L_i)$$

But this is not practical since we would need to solve a huge system every time.
The number of samples n is a hyperparameter called batch size.

Smaller batch size:
- Greater variance in the gradients, which leads to noisy updates
- slower convergence
- Needs more updates, but theoretically lower computational cost

Larger batch size:
- less variance in the gradients
- faster convergence
- higher computational cost, but we can use GPUs to parallelize the update

In practice, we want to choose the biggest possible batch size that can be parallelized on our GPUs in one backward pass. Often values such as 32, 64 up to 1024 are used, for large models the batch size could millions of samples.
Problems of Stochastic Gradient Descent:
- Gradient is scaled equally across all dimensions, cannot independently scale directions
- Learning rate needs to be conservative (small) to avoid divergence
- Small learning rate means slow convergence
- Finding the perfect learning rate is therefore difficult

Gradient Descent with Momentum

Instead of making many steps back and forth along the wrong dimensions, we would like to capture the momentum and move faster towards the optimum.
We calculate a velocitiy vector v:

$$v^{k+1} = \beta \cdot v^k - \alpha \cdot \nabla_\theta L(\theta^k)$$

which we use to update the weights:

$$\theta^{k+1} = \theta^k + v^{k+1}$$

The hyperparameter β is usually set to 0.9.
Essentially, we add an exponentially-weighted moving average of the gradient.
Gradient descent with momentum can potentially overcome local minima due to the momentum factor, which allows it to escape "small bumps".

Nesterov Momentum

A kind of look-ahead momentum.
First make a big jump in the direction of the previously accumulated gradient:

$$\widetilde{\theta}^{k+1} = \theta^k + \beta \cdot v^k$$

Then measure the gradient where you end up with the updated weights and make a correction:

$$v^{k+1} = \beta \cdot v^k - \alpha \cdot \nabla_\theta L(\widetilde{\theta}^{k+1})$$

$$\theta^{k+1} = \theta^k + v^{k+1}$$

RMSProp

Root Mean Squared Prop.
Divide the learning rate by an exponentially-decaying average of squared gradients.

$$s^{k+1} = \beta \cdot s^k + (1-\beta)[\nabla_\theta L \circ \nabla_\theta L]$$

$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{\nabla_\theta L}{\sqrt{s^{k+1}} + \epsilon}$$

where $\circ$ is element wise multiplication to square the gradients. β and ϵ are hyperparameters, usually set to $\beta = 0.9$ and $\epsilon = 10^{-8}$.
Essentially, we keep an moving average of squared gradients and adjust the weight updates by dividing by this average. The idea is that the learning rate can be adjusted for different gradients influenced by different weights, which is better than having a static learning rate.
This dampens the oscillations of gradient descent for high-variance directions.
It also can utilize a higher learning rate since it is less likely to diverge.

Adam

Adaptive Moment Estimation.
It combines Momentum and RMSProp.
The first momentum, the mean of gradients, is calculated as follows:

$$m^{k+1} = \beta \cdot m^k + (1-\beta_1)\nabla_\theta L(\theta^k)$$

The second momentum (RMSProp), the variance of gradients:

$$v^{k+1} = \beta_2 \cdot v^k + (1 - \beta_2)[\nabla_\theta L(\theta^k) \circ \nabla_\theta L(\theta^k)]$$

The updated weight are then:

$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{m^{k+1}}{\sqrt{v^{k+1}} + \epsilon}$$

Notice that at $k = 0$, we use $m^0 = 0$ and $v^0 = 0$. This leads to a bias towards 0. We therefore need bias-corrected moment updates:

$$\hat{m}^{k+1} = \frac{m^{k+1}}{1 - \beta_1^{k+1}}$$

and

$$\hat{v}^{k+1} = \frac{v^{k+1}}{1 - \beta_2^{k+1}}$$

Adam is the standard choice for neural networks today.

Second Order Optimization Methods

Newton's Method:
- Find a global minima by using the second order derivative.
- Theoretically faster and better (uses less steps) at finding a global minimum than gradient descent.
- In reality, too complex to compute second order derivative.

BFGS and L-BFGS:
- Broyden-Fletcher-Goldfarb-Shanno algorithm.
- quasi Newton method.
- uses an approximation of the inverse of the Hessian.
- reduces complexity from $O(n^3)$ to $O(n^2)$.

Other second-order methods: Gauss-Newton (GN), Levenberg, Levenberg-Marquardt (LM).

The main problem of these methods is, that you need to be able to do a full batch update, meaning update the network weights at once using the full dataset. This is practically never the case in deep learning, which is why gradient descent methods on minibatches is the way to go.

Regularization

Prevent overfitting with regularization.
Add regularization to the loss:

$$L'(y, \hat{y}; \theta) = L'(y, \hat{y}; \theta) + \lambda R(\theta)$$

L2 Regularization:

$$R(\theta) = \sum_{i=1}^{n} \theta_i^2$$

- Is lower for smaller, distributed values.
- Enforces that weights have similar values.
- Model will take all information into account to make decisions.

L1 Regularization:

$$R(\theta) = \sum_{i=1}^{n} |\theta_i|$$

- Is overall lower than L2 with some higher weight values.
- Enforces sparsity.
- Model will focus attention on a few key features.

Regularization coefficient λ sets the level of regularization (higher λ means more regularization).
General effect of regularization: Lower validation error, but increased training error.