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
θ = 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).

Activation Functions

Sigmoid:

$$\sigma(x) = \frac{1}{1 + e^x}$$

Disadvantages:
- Vanishing gradient problem: gradients quickly approach 0
- The output is not zero-centered

TanH:

$$tanh(x)$$

Zero-centered, but still the vanishing gradient problem.
ReLU:

$$ReLU(x) = max(0, x)$$

Advantages:
- In practice, faster convergence than sigmoid/tanh
- No vanishing gradient problem, large and consistent gradients
- Computationally efficient

Disadvantages:
- Blows up output
- Dead neurons

Leaky ReLU:

$$LReLU(x) = max(0.01, x)$$

- Like ReLU, but does not cause dead neurons

Parametric ReLU:

$$PReLU(x) = max(\alpha x, x)$$

- One more parameter for backpropagation
- Also does not die

Maxout:

$$Maxout(x) = max(w_1^T x + b_1, w_2^T x + b_2)$$

- Returns the maximum of the inputs
- Generalization of the ReLU function

Output Functions

Output function is the last activation function of the network, which represents the network output.

Binary classification: Sigmoid

$$\sigma(x) = \frac{1}{1 + e^x}$$

The resulting value is between 0 and 1 which can be seen as a probability of a class outcome.

Multi-class classification: Softmax

$$\sigma(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$$

The exponential operation makes sure the probabilities are above 0. The normalization makes sure all resulting probabilities sum up to one.

Loss Functions

Measure the goodness of the prediction.

Large loss indicates bad performance.

Choice of loss function depends on problem.

Regression Loss

L1 Loss:

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

- Sum of absolute differences
- Robust, not prone to outliers
- Costly to optimize (not differentiable)
- Optimum is the median

MSE Loss (L2 Loss):

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

- Sum of squared differences
- Prone to outliers
- Compute efficient (differentiable)
- Optimum is mean

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: $||x||_2^2 = \sum x_i^2$
- Similar to L2 norm
- 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.
Hinge Loss (multi-class classification):

$$L_i = \sum_{k \neq y_i} max(0, s_k - s_{y_i} + 1)$$

- Hinge loss quickly saturates at 0 loss for good predicitions, which is not useful for neural network backpropagation. Cross entropy loss on the other hand always wants to improve, the loss is never 0.

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.

Bagging and Ensemble Methods

Train multiple models and average their result. Use different optimization methods or loss functions. Bagging: Use k different datasets.

Dropout

Disable a random set of neurons, typically 50%. Dropout reduces co-adaption between neurons. This is similar to training an ensemble of models but with shared paramters. It also can be seen as loosely equivalent to L2 regularization.

At test time and inference time, all neurons are turned on.

Disadvantages:

- Need more training time (around 1.5 times longer).
- Does not work well with batch normalization.

Batch normalization

Batch normalization:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

Second step, to allow the network to change the range:

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

where γ and β are learnable parameters.
This re-centers and re-scales the dimensions by normalizing the mean and variance. Batch Normalization is employed after the linear transformation but before the activation. Important is that Batch normalization treats dimensions k separatly. It is empirically shown that this works. On the other hand, layer normalization normalizes over dimensions instead of over batches. During testing and inference, we cannot calculate the mean and variance of a minibatch, therefore we compute an exponentially weighted average across training batches.
Internal Covariate Shift: The change of the distributions of the activations due to a change in parameters of previous layers. Batch Normalization was believed to help mitigate Internal Covariate Shift. Some scholars believe it instead helps smooth the loss function. Its much easier to train larger networks with batch normalization.

Training

Learning Rate

When far away from the minimum during trainig, we need high learning rate. But when closer to the minimum, we want a small learning rate to not overshoot it.
We can use learning rate decay to lower the learning rate over epochs (one epoch = one complete pass over the training samples):

$$\alpha = \frac{1}{1 + t \cdot epoch} \cdot \alpha_0$$

where α_0 could be 0.1 and the decay rate t could be 1.0.
Other options on how to lower the learning rate:
- Step decay (only ever n steps): $\alpha = \alpha - t \cdot \alpha$.
- Exponential decay: $\alpha = t^{epoch} \cdot \alpha_0$
- $\alpha = \frac{t}{\sqrt{epoch}} \cdot \alpha_0$
- Or set the learning rate manually after every n epochs.

Data Split

Training set:

- Used for training the network
Validation set:
- Check generalization progress
- Hyperparameter Optimization
Test set:
- only for the very end, assesing the performance of your network
- never touch during development or training
Typical splits:
- Train: 60%, Validation: 20%, Test: 20%
- Train: 80%, Validation: 10%, Test: 10%
Cross Validation:
- For each iteration, take a different part of the training data as validation data

Over- and Underfitting

Training error high?
- Bigger model
- Train longer
- Different architecture
Validation error high (Overfitting)?
- More data
- Regularization
- Different architecture

Hyperparameter Tuning

Methods to find the best hyperparameters:
- Manual search, most common
- Grid search: Define ranges for all parameters and select points.
- Random search: Like grid search, but picks points at random in the predefined ranges
How to start:
- Start with a single training sample, check if output is corrected, accuracy should be 100- Increase to a handful of samples
- Move from overfitting to more samples, should lead to generalization

Data Augmentation

A classifier has to be invariant to a wide variety of transformations.

Synthesize data using plausible transformations:
- Brightness and contrast changes
- Random crops (parts of the image)

Weight Initialization

All weights zero:
- The hidden units compute all the same value, gradients are the same
Small random numbers (0 mean, 0.01 standard deviation):
- Vanishing gradients especially with tanh or sigmoid activations
Big random numbers (0 mean, 1 standard deviation):
- Outputs saturated to -1 or 0 and 1 with tanh or sigmoid, causing vanishing gradients
Xavier initialization:
- Initialize with 0 mean and $Var(w) = \frac{1}{n}$ and bias 0 at each layer, where n is the number of input neurons.
- This leads to good distributions.
- When using ReLU, it kills half the neurons. Therefore, we need to double the variance: $Var(x) = \frac{2}{n}$.
- Xavier is proven to be the best initialization method.

Convolutional Neural Networks

Convolutional Layer

Learnable convolutional filters. Extremely important for computer vision tasks since networks composed of fully connected linear layers would require too many parameters.
A 2D convolutional layer consists of:
- Number of filters
- Filter size $F \times F$, the depth is given by the number of channels in the input.
- The stride S, meaning the movement of the filter after each convolution
- The padding P, which pads the input with zeros.
We can calculate the output dimension by using the input size $N \times N$:

$$\left(\left\lfloor \frac{N + 2P - F}{S} \right\rfloor + 1\right) \times \left(\left\lfloor \frac{N + 2P - F}{S} \right\rfloor + 1\right)$$

To achieve the same output dimension as the input dimension, use padding $P = \frac{F-1}{2}$.
The weight tensor of the convolutional layer has shape $(channels, filters, F, F)$.

Pooling Layer

Select a ceratain feature in a region. Max Pooling selectes the max value of a region in the input. Average Pooling computes the average of a region.
Dimensions:
- Input $W_{in} \times H_{in} \times D_{in}$
- Spacial filter extent F
- Stride S
- Output: $W_{out} = \frac{W_{in}-F}{S} + 1$, $H_{out} = \frac{H_{in}-F}{S} + 1$, $D_{out} = D_{in}$

Does not contain any parameters, its just a fixed function.

Other Layers

1x1 Convolution: Useful for shrinking the number of channels.
Inception Layer: Use multiple different convolutions in one layer and concatenate the outputs (as done in GoogLeNet/XceptionNet).

Common Architectures

LeNet/AlexNet/VGG: Conv ¿ Pool ¿ Conv ¿ Pool ¿ Conv ¿ FC (Fully Connected).
As we go deeper in the network, the width and height of the image decreasees, but the number of filters (the depth) increases.
Larger networks makes training harder, vanishing or exploding gradient become more common. This was a bottleneck, until residual blocks were invented.
Residual Block: Two layers, and the input to the first layer is added to the output of the second layer before the activation. Residuals help mitigate the vanishing gradient problem, since on the gradient computation the added summand will make gradients not go towards 0. In the forward pass, it allows for signal propagation and therefore includes the filter information from the previous block in the activation of the next block.

Upsampling

Upsampling for CNNs: Turn a latent representation of the image with increased depth back to a bigger image with less depth.
Two types of upsampling:
- Interpolation: Interpolate the missing pixels (Bilinear interpolation, Bicubic interpolation)
- Transposed Convolution/Up-Convolution: A Convolutional Layer that increases the output size.

U-Net: Encoder-Decoder like CNN with Skip Connections.
Encoder (Contraction Path): Typical CNN architecture; Convolutions, ReLU, MaxPool; decreasing width/height, increasing depth.
Decoder (Expansion Path): 2x2 up-convolutions that increase the height and width but decrease the depth; skip connections from the encoder.
The result is an image with similar size to the input, which allows for tasks that require class labels on each pixel such as image segmentation.

Transfer Learning

Re-use pre-trained models. For example, take a big CNN model and only replace the last fully connected layer with another fully connected layer that has the amount of output classes for your task. Train only this layer, freeze the remaining layers in the network. If you dataset is big enough, you can train all layers with a low learning rate.
Transfer learning is useful if the dataset of the original model and your dataset have the same input, when there is way more data for the original task and when features learned for the original task could be useful for the second task.
Representation Learning: Learn deep representation for a variety of tasks. For example, a representation learning network could try to maximize the similarity between the same image transformed in different ways. The acquired representation in the network are then used in the downstream task.

Recurrent Neural Networks

Process sequential data.
Many types of RNNs: Many-to-one, one-to-many, many-to-many.
Formal definition of a RNN hidden state:

$$A_t = \theta_c A_{t-1} + \theta_x x_t$$

where $\theta_c A_{t-1}$ is the output of the previous hidden state and $\theta_x x - t$ is the input to this hidden state. The weights of this hidden unit are shared across all hidden units in the layer. The weights are only different between layers. The reason for this is so the model can generalize to different sequence lengths. Also sometimes the same operation should be performed even if the order in the sentence changes (as sometimes in NLP). A practical reason was also that less parameters make the training more feasable.

A big problem with RNNs are long term dependencies in sentences. Large weights for that input (eigenvalue > 1) lead to exploding gradients and small weights (eigenvalue < 1) lead to vanishing gradients.

Long-Short Term Memory Units

RNN, but with additional ingredients to each hidden state:
- Cell: Transports the information from the previous unit to the next unit.
- Forget gate: Takes the input and previous output, applies linear transformation and sigmoid, and element-wise multiplies the output with the cell state. Intuition: The sigmoid output 0 means forget and 1 means keep. This decides if the cell state should be kept or forgotten.
- Input gate: Takes the input and previous output, applies linear transformation and sigmoid, take the current state and apply tanh, combine these two with pointwise multiplication and add it onto the cell state. Intuition: Decides which values should be updated on the cell state.
- Output gate: