Neural Networks

Neural networks are a class of **machine learning** models inspired by the **brain**. They are non-linear models that can be used for **regression** and **classification**.

Definition

- Inspired by biological **neurons**;
- An **artificial neuron** is a **function** $f : \mathbb{R}^d \to \mathbb{R}$;
  - It receives a **vector** of **inputs** $x \in \mathbb{R}^d$, **weights** $w \in \mathbb{R}^d$ and a **bias** $b \in \mathbb{R}$:

$$z(x) = w^T x + b = \sum_{i=1}^{d} w_i x_i + b$$

The **activation function** g is a function that **transforms** the **output** of the **neuron**:

$$h(x) = g(z(x)) = g(w^T x + b), whereg : \mathbb{R} \to \mathbb{R}$$

Activation Functions

The typical **activation functions** are:

- **Linear**: $g(z) = z$;
  - No **squashing**;
- **Sigmoid**: $g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$;
  - Squashes z to $[0, 1]$;
  - Output can be interpreted as a **probability**;
  - Positive, bounded and strictly increasing;

- **Hyperbolic tangent**: $g(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$;
  - Squashes z to $[-1, 1]$;
  - Bounded and strictly increasing;
- **Rectified Linear Unit (ReLU)**: $g(z) = relu(z) = max(0, z)$;
  - Non-negative, increasing but **not upper bounded**;
  - Not differentiable at $z = 0$;
  - Leads to neurons with **sparse activities-**

Later, we will see:

- **Softmax**: $g(z) = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}}$;
- **Sparsemax**: $g(z) = argmin_{p \in \Delta_d} ||p - z||_2^2$;
- **Max-pooling**: $g(z) = max(z)$.

Feedforward Neural Networks

- To solve non-linear problems, we can **stack** several **neurons**;
- **Multi-layer neural networks**: use intermediate layers between the input and the output layers;
- Each hidden layer computes a representation of the input and propagates it to the next layer - **feedforward neural networks**.

Single Hidden Layer

- Consider a task that involves several **inputs** $x \in \mathbb{R}^d$ and a **single output** $y \in 0, 1$;
- Include an **intermediate layer** of K **hidden units** ($h \in \mathbb{R}^K$) between the input and the output layers;
- **Hidden layer pre-activation**: $z(x) = W^(1)x + b^(1)$, with $W^(1) \in \mathbb{R}^{K \times d}$ and $b^(1) \in \mathbb{R}^K$;
- **Hidden layer activation**: $h(x) = g(z(x))$, with g being the **activation function** $g : \mathbb{R}^{\mathbb{K}} \rightarrow \mathbb{R}^{\mathbb{K}}$;
- **Output layer activation**: $y(x) = \sigma(w^(2)h(x) + b^(2))$, with $w^(2) \in \mathbb{R}^{\mathbb{K}}$ and $b^(2) \in \mathbb{R}$.

Multiple Classes

- **Multiple output units**, one for each class;
- Each output estimates the conditional probability of the input belonging to that class - **softmax** activation function:

$$o(z) = softmax(z) = \begin{bmatrix} \frac{e^{z_1}}{\sum_{j=1}^{d} e^{z_j}} \\ \vdots \\ \frac{e^{z_d}}{\sum_{j=1}^{d} e^{z_j}} \end{bmatrix}$$

- **Hidden layer pre-activation**: $z^(l)(x) = W^(l)h^(l-1)(x) + b^(l)$, with $W^(l) \in \mathbb{R}^{K^(l) \times K^(l-1)}$ and $b^(l) \in \mathbb{R}^{K^(l)}$;
- **Hidden layer activation**: $h^(l)(x) = g(z^(l)(x))$, with g being the **activation function** $g : \mathbb{R}^{\mathbb{K}^(\mathbb{l})} \rightarrow \mathbb{R}^{\mathbb{K}^(\mathbb{l})}$;
- **Output layer activation**: $y(x) = softmax(w^(L)h^(L-1)(x) + b^(L))$, with $w^(L) \in \mathbb{R}^{\mathbb{K}^(\mathbb{L})}$ and $b^(L) \in \mathbb{R}$.

Universal Approximation Theorem

> *A Neural Network with a **single hidden layer** and a **linear output layer** can **approximate any continuous function** to **arbitrary accuracy** if the **hidden layer has enough units**.* - Cybenko, 1989

- **Deeper networks** (with more hidden layers) can **approximate functions more efficiently**;
- The number of linear regions carved out by a deep neural network with D inputs, depth L and K hidden units per layer is $O((\frac{K}{D})^{D(L-1)}K^D)$;
- So neural networks can **approximate any function** with a **single hidden layer**: we only need to find the **right parameters - training**.

Training Neural Networks

Training a neural network means **finding the right parameters** for the **weights** and **biases** of the **neurons**, learning them from data - samples of **inputs** and **outputs**.

Empirical Risk Minimization

- Goal: choose parameters $\theta := (W^{(l)}, b^{(l)})_{l=1}^{L+1}$ that **minimize the empirical risk**:

$$\mathcal{L}(\theta) := \lambda\Omega(\theta) + \frac{1}{N}\sum_{n=1}^{N} L(f(x_i;\theta), y_i)$$

- $x_i, y_i{}_{i=1}^{N}$ is the **training set**;
- $L(f(x_i;\theta), y_i)$ is the **loss function**;
- $\omega(\theta)$ is the **regularization term**;
- λ is the **regularization constant**.
- **Gradient descent** is **too slow**, because it requires a full pass over the data to update the weights;
- **Stochastic gradient descent** is **faster**, because it updates the weights after each sample;
- **Mini-batch stochastic gradient descent** is **even faster**, because it updates the weights after each mini-batch of samples;
  - A **mini-batch** is a **subset** of the **training set**, $j_1, \dots, j_B$, with B samples, $B \ll N$:

$$\nabla_\theta \mathcal{L}(\theta) = \lambda \nabla_\theta \omega(\theta) + \frac{1}{B}\sum_{i=1}^{B} \nabla_\theta L(f(x_{j_i};\theta), y_{j_i})$$

The weights are updated as follows:

$$\theta^{(t+1)} = \theta^{(t)} - \eta\frac{1}{B}\sum_{i=1}^{B} \nabla_\theta L_{j_i}(\theta^{(t)})$$

- **Loss function** L should match as possible what we want to **optimize**;
  - Should be well-behaved - continuous and smooth;
  - **Squared loss** is a good choice for **regression**;
  - **Cross-entropy** loss is a good choice for **multi-class classification**: $L(f(x;\theta), y) = -log((softmax(f(x;\theta)))_y)$;
  - **Sparsemax** loss is a good choice for **multi-class and multi-label classification**.

Backpropagation

We need to find a **procedure** to **compute the gradients** of the **loss function** with respect to the **weights** and **biases** of the **neurons**: $\nabla_\theta L(f(x;\theta), y)$.

- The **gradient backpropagation algorithm** is a **recursive algorithm** that computes the **gradients** of the **loss function** with respect to the **weights** and **biases** of the **neurons**;
- It is based on the **chain rule** of **calculus**:

$$h(x) = f(g(x)) \Rightarrow \frac{dh}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

Automatic Differentiation

- **Automatic differentiation** is a **technique** for **computing derivatives** of **functions**;
- Forward propagation can be represented as a **computation graph** - a **directed acyclic graph** (DAG) that represents the **computation** of the **function**;
  - Each box can be an object with a `fprop` method that computes the **forward pass**;
  - Calling the `fprop` method of each box in the **topological order** of the graph computes the **forward pass**;
- **Backpropagation** is also implemented as a **computation graph** - a **directed acyclic graph** (DAG) that represents the **computation** of the **gradients**;
  - Each box can be an object with a `bprop` method that computes the **loss gradient** w.r.t. its parents, given the **loss gradient** w.r.t. to the output of the box;
  - Calling the `bprop` method of each box in the **reverse topological order** of the graph computes the **backward pass**.

There are several **Autodiff** strategies:

- **Symbol-to-Number Differentiation**
  - Take a computational graph and numerical inputs;
  - Returns a set of numerical outputs describing the gradient at those inputs;
  - **Advantage**: simpler to implement and debug;

- **Disadvantage**: only works for first-order derivatives;
- **Example**: Caffe, Torch, PyTorch, ...

- **Symbol-to-Symbol Differentiation**
  - Take a computational graph and add additional nodes to the graph that provide a symbolic description of the gradient;
  - **Advantage**: works for higher-order derivatives;
  - **Disadvantage**: more complex to implement and debug;
  - **Example**: Theano, TensorFlow, ...

Regularization

Recall the **empirical risk minimization** problem:

$$\mathcal{L}(\theta) := \lambda\Omega(\theta) + \frac{1}{N}\sum_{n=1}^{N} L(f(x_i;\theta), y_i)$$

- It remains to define the **regularizer** $\Omega(\theta)$ and its **gradient** $\nabla_\theta \Omega(\theta)$;

$\updownarrow_2$ Regularization

- Only the **weights** are regularized: $\Omega(\theta) = \sum_{l=1}^{L+1} ||W^{(l)}||^2$;
- Equivalent to **Gaussian prior** on the weights;
- **Gradient of the regularizer** w.r.t. the **weights**: $\nabla_{W^{(l)}}\Omega(\theta) = W^{(l)}$;
- **Weight decay effect**: the **weights** are **shrunk** towards **zero**: $W^{(l)} \leftarrow (1-\eta\lambda)W^{(l)}$;

$\updownarrow_1$ Regularization

- Only the **weights** are regularized: $\Omega(\theta) = \sum_l ||W^{(l)}||_1 = \sum_l \sum_{i,j} |W_{ij}^{(l)}|$;
- Equivalent to **Laplace prior** on the weights;
- **Gradient of the regularizer** w.r.t. the **weights**: $\nabla_{W^{(l)}}\Omega(\theta) = sign(W^{(l)})$;
- Promotes **sparsity** in the weights: **many weights** are **zeroed out**.

Dropout Regularization

- During training, **randomly drop** some of the **neurons** in the **hidden layers**;
- Each hidden unit output is set to zero with probability p - this prevents hidden units to **co-adapt** to each other, forcing them to be more generally useful;
- At test time, keep all units with the **outputs multiplied by** $1 - p$;
- Usually implemented using **random binary masks**;
- The hidden layer activations becomes: $h^{(\ddagger)}(x) = g(z^{(\ddagger)}(x)) \odot m^{(\ddagger)}$, where $m^{(\ddagger)} \in \{0, 1\}^{K^{(\ddagger)}}$ is a **random binary mask** with p probability of being 1;

Tricks of the Trade

Initialization

- **Biases**: set to zero;
- **Weights**:
  - Cannot be zero with **tanh** activation function;
  - Cannot be all the same value - use **random initialization** - **Gaussian** or **uniform**;
  - For **ReLU**, the mean should be a small positive number;
  - Variance cannot be too high;

Training, Validation and Test Sets

- **Training set**: used to **train** the model;
- **Validation set**: used to **tune** the **hyperparameters** (e.g. learning rate, regularization constant, etc.);
  - **Grid search** specify a set of values to test for each hyperparameter, and try all combinations;
  - **Random search** specify a distribution for each hyperparameter, and sample from it;
  - **Bayesian optimization** specify a prior distribution for each hyperparameter, and update it after each experiment;

- **Test set**: used to **evaluate** the **final model**.

  Early stopping is a **regularization technique** that stops training when the **validation error** starts to **increase**, in order to prevent **overfitting**.

Input Normalization

- Subtract the mean and divide by the standard deviation;
- It makes each input dimension have **zero mean** and **unit variance**;
- It **speeds up** the **training**;
- Does not work for **sparse data**;

Decaying the Learning Rate

- **Learning rate** η is a **hyperparameter** that controls the **step size** in the **gradient descent** algorithm;
- In SGD, as we get closer to the minimum, we want to **reduce the step size**:
  - Start with a large learning rate (e.g. 0.1);
  - Keep it fixed while the **validation error** is **decreasing**;
  - Divide by 2 and repeat.

Mini-Batches

- Instead of updating after a single sample, update after a **mini-batch** of samples (e.g. $50 - 200$ samples), and compute the average gradient for the entire mini-batch;
- Less noisy than SGD;
- Can leverage matrix-matrix computations.

Gradient Checking

- If the training loss is **not decreasing**, there might be a **bug** in the **gradient computation**;
- To debug, we can compute the **numerical gradient** and compare it with the **analytical gradient**:

$$\frac{\partial f}{\partial x} \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

Better Optimization

There are several improvements to basic **gradient descent** and **stochastic gradient descent**:

- **Momentum**;
- **Adaptive gradient** (AdaGrad);
- **Root mean square propagation** (RMSProp);
- **Adaptive moment estimation** (Adam).

Momentum

- **Momentum** is a **technique** that **accelerates gradient descent** in the **relevant direction** and **dampens** oscillations;
- It means: remember the **previous gradients** and use them to **update** the **current gradient**: $\theta_t = \theta_{t-1} - \alpha_t g(\theta_{t-1}) + \gamma_t(\theta_{t-1} - \theta_{t-2})$;
  - $g(\theta_t)$ is the **gradient estimate** at time t;
- **Advantages**: reduces the update in irrelevant directions and accelerates the update in relevant directions.

Adaptive Gradient (AdaGrad)

- **AdaGrad** is a **technique** that **adapts** the **learning rate** to the **parameters**, performing **smaller updates** for **frequent** parameters and **larger updates** for **infrequent** parameters;
- Scale the update of each component (ϵ for numerical stability): $\theta_{j,t} = \theta_{j,t-1} - \frac{\alpha}{\sqrt{G_{j,t-1}+\epsilon}} g_j(\theta_{t-1})$;
  - $G_{j,t}$ is the **sum of squares of the gradients** w.r.t. θ_j up to time t: $G_{j,t} = \sum_{i=1}^{t} g_{j,i}^2$;
- **Advantages**: robust to choice of α and **learning rate decay**;
- **Disadvantages**: step size vanishes, because $G_{j,t}$ is monotonically increasing.

Root Mean Square Propagation (RMSProp)

- **RMSProp** addresses the **vanishing learning issue**;
- Same scaled update for each component: $\theta_{j,t} = \theta_{j,t-1} - \frac{\alpha}{\sqrt{G_{j,t-1}+\epsilon}} g_j(\theta_{t-1})$;
- Use a **forgetting factor** γ to **decay** the **sum of squares of the gradients** (typically $\gamma = 0.9$): $G_{j,t} = \gamma G_{j,t-1} + (1-\gamma) g_{j,t}^2$ - now the **sum of squares of the gradients** is **decaying**.

Adaptive Moment Estimation (Adam)

- **Adam** is a **combination** of **momentum** and **RMSProp**;
- Separate moving averages of gradient and squared gradient;
- **Bias correction** is used to initialize the moving averages to zero;
- **Hyperparameters**: α, β_1, β_2, ϵ;
- **Advantages**: computationally efficient, low memory requirements, well suited for problems with large datasets and/or parameters;
- **Disadvantages**: possible convergence issues and noisy gradient estimates.