Course Notes on Neural Networks

Course: Computational Intelligence (TI2736-A)
Delft University of Technology
Author: Thomas Moerland

Contents

1 Introduction

For these course notes we focus on *parametric, supervised* learning.

- *Supervised* learning describes the problem type. In supervised learning, we intend to learn a (vector) function mapping $f : \mathbf{x} \rightarrow \mathbf{y}$, given a dataset of observed examples $(\mathbf{x}_i, \mathbf{y}_i)$, $i = 1...N$. The learned function is intended to - for a new datapoint $\mathbf{x}$ - make a *prediction* $\hat{\mathbf{y}}$ (say y 'hat', as it not unobserved).
- *Parametric* refers to the solution approach. Parametric models fix the *number* of elements in the model in advance (i.e., the model is independent of the size of the dataset). The parameters, denoted by θ, determine the function that maps $\mathbf{x}$ to $\mathbf{y}$.

We may either write $f(x;\theta)$ or $f_\theta(x)$ for the parametrized function that maps $\mathbf{x}$ to $\mathbf{y}$. Parametric supervised learning is probably the largest class of machine learning approaches. The learning approach to this problem consists of three fundamental concepts:

1. **A parametric model architecture**
2. **A loss/objective function**
3. **An optimization method**

Deep neural networks (NN) are a particular form of a parametric model architecture, with associated specific optimization methods. In these notes we cover these three topics for the NN case. We first summarize notation for reference.

2 Notation

$\mathbf{x}$	Input/independent variable)
y	Output/target/dependent variable
$\mathcal{D} = \{\mathbf{x_1}, y_1, \mathbf{x_2}...\mathbf{x_N}, y_N\}$	Dataset with N examples
$\hat{y}$	Predicted value of y
$\theta = \{W^{(i)}, b^{(i)}\}$	Model parameter set, for network layers $i = 1...d$
$f(\mathbf{x};\theta) = f_\theta(\mathbf{x})$	Neural network with point prediction (parametrized by θ)
$p(y\|\mathbf{x})$	Conditional probability distribution
$p(y\|\mathbf{x};\theta) = p_\theta(y\|\mathbf{x})$	Neural network with probability distribution as output
$\mathbf{W}^{(n)}$	Weight matrix in n-th layer
$w_{j,i}^{(n)}$	Weight between i-th node in layer $n-1$ to j-th node in layer n
$\mathbf{b}^{(n)}$	Bias vector in n-th layer
$g^{(n)}(\cdot)$	Non-linearity / Activation function in the n-th layer
$\mathbf{h}^{(n)}$	Hidden units in n-th layer
$\mathbf{z}^{(n)}$	Linear activation in the n-th layer, before applying $g(\cdot)$
$\mathcal{L}(\theta\|y, \mathbf{x})$	Likelihood/loss function
α	Learning rate

3 Model architecture

A neural network describes a type of parametric function class. It takes as input some (possibly structured) data vector $\mathbf{x} \in \mathbb{R}^m$ of dimension m, and transforms this according to parameters θ to an output $\mathbf{y}$ (more on different output types in the loss function section). The important distinction between data $(\mathbf{x}, \mathbf{y})$ and parameters θ is that the data is observed (and therefore fixed), while the parameters will be learned/changed. We do not know the correct parameter setting in advance. Therefore, the parametric model basically describes a *space* of functions, i.e., for each parameter setting we get a different function, and our optimization will search over this space of functions. The way in which we combine our data and the parameters is the model architecture, and depending on our choices the model function space will be larger or smaller. We usually call the size of the function space that a particular class can approximate the *capacity* of the model. Deep neural networks are especially popular because they are high-capacity models that can handle high-dimensional input data x.

Our network outputs a prediction for the target, which we denote by $\hat{y}$. The overall network is then specified by:

$$\hat{y} = f(\mathbf{x}; \theta) = f_\theta(\mathbf{x})$$

3.1 Two-layer neural network

We will now discuss the standard two-layer feedforward network (= multi-layer perceptron (MLP)), which is a particular type of parametric model architecture. The term feedforward refers to the fact that there are no feedback connections in the network, i.e., we feed in x and the computation moves forward through the network to predict a $\hat{y}$. Networks usually consist of several layers, where each layer effectively performs a non-linear regression.

The linear transformation We first construct a linear transformation of the input x. We assume the input vector $x \in \mathbb{R}^m$ of size m maps to an output vector $z \in \mathbb{R}^n$ of size n. We then need a **weight matrix** of size $n \times m$ and a **bias vector** of size n to construct the linear transformation:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

This describes a set of n linear regressions, where the weight matrix and biases are the adjustable parameters (part of θ).

The activation function We then apply an element-wise non-linearity $g(\cdot)$ to the output of the linear part z. Element-wise means that the function $g(\cdot)$ is applied to every element in the vector $\mathbf{z}$. The main requirement is that the non-linearity should be differentiable (although the ReLu actually is not differentiable at 0, but we ignore this issue here). We list and visualize a few common choices for the activation function:

1. Rectifier linear unit (**ReLu**): $g(z) = \begin{cases} 0, & \text{if } z < 0 \\ z, & \text{if } z \geq 0 \end{cases}$
2. Exponential linear unit (**ELU**): $g(z) = \begin{cases} e^z - 1, & \text{if } z < 0 \\ z, & \text{if } z \geq 0 \end{cases}$
3. **Sigmoid**: $g(z) = \dfrac{1}{1 + e^{-z}}$
4. Hyperbolic tangent (**Tanh**): $g(z) = \tanh(z)$

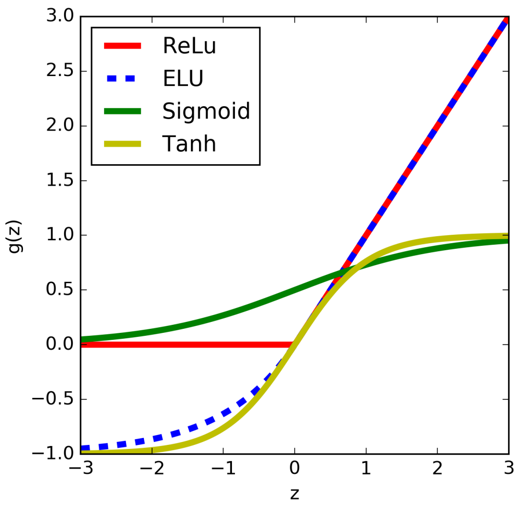

Figure 1. Activation functions.

Older work on neural networks mostly utilized sigmoid and tanh activations, but more recent work shows better performance for partially linear units (Relu and Elu). Their main benefit is that optimization turns out to be easiest in the linear setting, as gradients backpropagate most easily through linear layers (the gradient may vanish in 'flat' regions of the non-linearity). Moreover, it is interesting to note that ReLu units were actually developed from bio-inspiration. The underlying observations in neuroscience are that neurons are 1) sometimes completely inactive, 2) when active the output is usually proportional to the input and 3) most neurons are usually inactive (i.e. activations are **sparse**).

Multiple layers and stacking We may compose the linear transformation and the non-linearity into one network **layer**:

$$\mathbf{h} = g(\mathbf{W}\mathbf{x} + b)$$

The key idea of neural networks is to repeatedly apply such layer-wise transformations, known as layer **stacking**. For a two-layer neural network, the transformations are:

$$\hat{y} = f^{(2)}(f^{(1)}(\mathbf{x}))$$

where we use superscripts $(\cdot)$ to identify the layer number. The first layer of this network computes:

$$\mathbf{h}^{(1)} = f^{(1)}(\mathbf{x}|\theta) = g^{(1)}(\mathbf{W}^{(1)}\mathbf{x} + b^{(1)})$$

The **hidden layer** h is then fed into the next layer.

$$\hat{y} = f^{(2)}(\mathbf{h}^{(\mathbf{1})}|\theta) = \mathbf{W}^{(2)}\mathbf{h}^{(\mathbf{1})} + \mathbf{b}^{(2)}$$

Because this is the last layer, we do not apply a non-linearity $g(\cdot)$ to the output. This ensures that we can predict a value for $\hat{y}$ on the entire real line (else we would restrict the values we can predict). Depending on the type of y variable we may add some other non-linear function, such as the softmax function in case of classification, or the exp function if we want to predict a standard deviation. You will read more about this in the section about **loss functions**. For now, we simply ignore the non-linearity in the last layer.

We can also write the network as one transformation:

$$\hat{y} = f_\theta(\mathbf{x}) = \mathbf{W}^{(2)}g^{(1)}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)}$$

In this example, the trainable parameters are given by $\theta = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)}\}$. Given a parameter setting of θ, we will get a prediction $\hat{y}$ for each value of $\mathbf{x}$ that we feed in. In the next section we describe how to construct a loss function, which measures how well the current prediction matches the true target in the dataset, and serves as an objective during training.

4 Loss Functions

The type of loss function strongly depends on the type of target variable y. If the target is continuous, then we call it **regression**. In that we case we do **not** need a non-linearity in the last layer, to ensure that we can predict on the entire real line. On the contrary, when the target is discrete, we call it **classification**, and we usually need a non-linearity to decide between the classes. The second import distinction between loss types are the **deterministic** versus **probabilistic** loss functions.

4.1 Regression

Mean-Squared Error We first focus on the probably best-known loss function, the mean-squared error (MSE). This is a deterministic regression loss. The loss function takes the squared error between the prediction of the network, $\hat{y}_i = f(\mathbf{x}_i;\theta)$, and the true observation in the dataset, y_i (per datapoint i):

$$\mathcal{L}(\theta|y,\mathbf{x}) = \mathbb{E}_{\mathcal{D}}\left[\left(f(\mathbf{x};\theta) - y\right)^2\right] = \frac{1}{N}\sum_{i=1}^{N}\left(f(\mathbf{x}_i;\theta) - y_i\right)^2$$

The squaring of the error ensure that we penalize both positive and negative errors (in equal amount). As an alternative, we could also use the absolute error, $|f(\mathbf{x}_i;\theta) - y_i|$, but the squared error is more easily to differentiate.

Maximum Likelihood The main idea of probabilistic losses is to use the function approximator to not predict $\hat{y}$ directly, but rather to predict the *parameters of a probability distribution* from which the observed y is a sample. The benefit is that we can model stochastic outputs and observation noise, and it also gives a principled way to construct loss functions.

We first note that the dataset $\mathcal{D}$ is actually a representation of a true data distribution p_{data} in the real world (from which we collected the data). We call the observed dataset the it empirical distribution, and denote it by $\hat{p}_{data}$. We should clearly discriminate this distribution from the *model* distribution $p_{model} = p_\theta(y|x)$, which is the probability distribution that our neural network predicts (for the probabilistic loss setting). We of course want our neural network to match the true (unknown) data distribution p_{data} as closely as possible. A popular idea to achieve this is to maximize the probability (likelihood) of our data under the model, i.e.:

$$\mathrm{L} = \mathbb{E}_{(x,y)\sim\hat{p}_{data}}\left[p_\theta(y|x)\right]$$

where $\mathbb{E}_{(x,y)\sim\hat{p}_{data}}$ means the **expectation** over the dataset. The above objective is known as **maximum likelihood estimation (MLE)**. In the neural network context introduced before, this becomes (now writing dataset $\mathcal{D}$ for the empirical distribution again):

$$\mathrm{L}(\theta|y,\mathbf{x}) = \mathbb{E}_{\mathcal{D}}\left[p_\theta(y|x)\right] = \prod_{i=1}^{N} p_\theta(y_i|x_i)$$

Maximum Likelihood for Regression To construct a likelihood we first need to assume a distribution for our dependent variable. In case our y variable is continuous, the best known choice is to specify a Normal outcome distribution, $p_\theta(y|x) = \mathcal{N}_\theta(y|x)$. We use our neural network to predict a mean $\mu(x;\theta)$ and standard deviation $\sigma(x;\theta)$ (i.e., we specify a network with two output heads, one for the mean, and one for the standard deviation. Note that σ should be larger than zero by definition, so we usually add some non-linearity that will ensure this for the σ head, for example exp function). If we assume that all our observations are **i.i.d.** (independently identically distributed), the we may write the joint likelihood as the product of the likelihood over all individual datapoints:

$$\mathrm{L}(\theta|y, \mathbf{x}) = \prod_{i=1}^{N} \mathcal{N}(y_i|\mu(x;\theta), \sigma(x;\theta))$$

We usually log transform the above equation (the log function is monotone, so it does not change the optimum, and it has the nice property of changing the product in to a sum), and because we usually want to minimize a quantity we negate the objective. The loss function $\mathcal{L}$ then becomes the **negative log-likelihood (NLL)**:

$$\mathrm{NLL} = \mathcal{L}(\theta|y, \mathbf{x}) = -\sum_{i=1}^{N} \log \mathcal{N}(y_i|\mu(x;\theta), \sigma(x;\theta))$$

In most applications with a continuous outcome (regression) we ignore σ by putting it at $\sigma = 1$. It turns out that, when we ignore σ, the above NLL cost is actually equal to minimizing the mean-squared error (MSE) loss introduced above (the loss is differs by a constant, but the optimal parameters are the same). This makes a nice connection between MSE and MLE training for the regression case. We will not go into further detail here, but remember that the parameter setting that minimize the MSE loss (/objective) is equal to the parameters setting that maximizes the likelihood under a Normal distribution (when ignoring σ, i.e. for μ only).

4.2 Classification

Maximum Likelihood for discrete y Probability theory is also the usual path to specify a loss function for the classification setting. A probability distribution over k classes is given by k class probabilities under the constraint that $\sum_k p_k = 1$. Therefore, we will implement our neural network to predict k different values on the real line (i.e. with no non-linearity yet). We then apply the **softmax classifier**, which approximates the above class probabilities p_k. The last linear layer gives us a vector f of length k each containing the unnormalized log probability of each class. We define the **cross-entropy loss** as:

$$\mathcal{L}(\theta|y, \mathbf{x}) = -\sum_{i=1}^{N} \log p(y_i|x) = -\sum_{i=1}^{N} \log \left(\frac{e^{f_{y_i}(x)}}{\sum_j e^{f_j(x)}} \right)$$

where f_{y_i} is the network head belonging to the true data class (y_i), and j indexes over the entire vector f. Intuitively, we first take the exp of all output nodes to make the numbers ≥ 0 (they should become probabilities), and then we normalize the terms to sum up to 1 (by dividing over their sum) to make it a valid probability distribution. The loss term then consists of the probability of the correct class, which is the quantity that we want to maximize. Note that, apart from pushing the probability of the correct class up, this loss also tries to push the probability of the wrong class labels down (due to the normalization term in the denominator).

5 Gradient Descent

All optimization problems depart from a **loss function**, also known as the **objective**, **cost** or **error** function. We are generally looking for the parameter setting $\theta^\star$ that minimizes the objective:

$$\theta^\star = \arg\min_\theta \mathcal{L}(\theta|y, \mathbf{x})$$

Optimization methods frequently utilize the derivative of the loss function with respect to the parameters. When the function has multiple parameters (i.e., θ is usually a vector/matrix of inputs) then we need the partial derivatives $\frac{\partial \mathcal{L}}{\partial \theta_j}$, for the derivative with respect to the j-th element of θ. The **gradient**, denoted by $\nabla_\theta \mathcal{L}$, is the generalized notion of the derivative in the case where the input is higher-dimensional, i.e. the gradient is the vector/matrix with all the partial derivatives.

The main idea of gradient descent is to *iteratively move the parameters in the direction of the negative gradient.* This gives the following update rule:

$$\theta' = \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$$

where θ is the old parameter setting, θ' is the updated parameter setting, and α is a **learning rate**. The learning rate decides how far we move in the direction of the gradient per step, and is a very important hyperparameter in neural network training. It should not be too small (or you will never make any progress in the optimization space) nor too large (or you will jump too far and optimization will be unstable).

Note that the objective function is the context of (deep) non-linear neural networks is generally non-convex. This implies that we are not garuanteed to reach the global optimal, but usually have to settle for a local optimum. In practice, the optimization algorithms usually manage to find reasonably good local optima.

5.1 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is a variant of gradient descent that scales to larger datasets. As a concrete example, we will assume the MSE loss introduced above. The gradient is given by:

$$\nabla_\theta \mathcal{L}(\theta|y, \mathbf{x}) = \sum_{i=1}^{N} \nabla_\theta \Big(f(\mathbf{x_i}; \theta) - y_i \Big)^2$$

However, when the size N of the dataset $\mathcal{D}$ grows larger, then the above summation becomes very computationally expensive. Therefore, most practical algorithms *approximate* the above gradient from a much smaller sample. Such a subsample from the dataset is called a **minibatch** of size m(usual choices of m are 32 or 64). The SGD gradient then becomes:

$$\mathbf{grad} = \sum_{i=1}^{m} \nabla_\theta \Big(f(\mathbf{x_i};\theta) - y_i \Big)^2$$

It turns out that this algorithms works very well in practice. It may not reach the global optimum, or even a local one, but it is able to reduce the cost function to a low value in reasonable time. The main benefit of the algorithm is that it scales to large datasets (as we can keep m fixed to a small number, independent of dataset size N).

5.2 Backpropagation

To implement the SGD update rule, we need the partial derivative of the cost $\mathcal{L}$ with respect to every parameter in θ. We can efficiently calculate these derivatives with the **backpropagation** algorithm (better known as 'backprop'). Backpropagating effectively propagates the error signal derived from the loss function back through the network, to figure out in which direction each parameter should be changed to make the total loss smaller. The backpropagation algorithm is based on two concepts: a) the chain rule of Calculus and b) efficient storage of gradients.

a) The chain rule of Calculus As we saw in the previous section, neural networks are essentially a sequence of differentiable transformations. We first revisit the **chain rule** of Calculus, which gives us an expression for the derivative of a composed function. Let $z = f(x)$ and $h = g(z) = g(f(x))$, then the chain rule gives us an expression for the derivative of h with respect to x: $\frac{dh}{dx} = \frac{dh}{dz}\frac{dz}{dx}$. In words, we are 'chaining' the derivatives of each individual transformation in the sequence.

These ideas generalize to the vector variable case, where we see the partial derivatives appearing again. For some $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{z} \in \mathbb{R}^n$, with $h = g(\mathbf{z})$ and $z = f(\mathbf{x})$, we have:

$$\frac{dh}{d\mathbf{x}_i} = \sum_j \frac{\partial h}{\partial z_j} \frac{z_j}{x_i}$$

.

In words, we sum the gradient over all the paths through the hidden variables z that reach the variable x_i. Note that the $\frac{\partial \mathbf{z}}{\partial \mathbf{x}}$ terms together form a matrix of first derivatives, known as the **Jacobian** (not to be confused with the Hessian, which is a matrix of second derivatives). In neural networks, we usually manipulate matrices of more than 2 dimensions, better known as **tensors**. The above equations equally apply to the case of tensors.

b) Efficiently computing the gradient The above rule gives us an expression for the gradient of the loss with respect to every parameter in our model $\frac{\partial \mathcal{L}}{\partial \theta_i}$. However, because we need to evaluate all paths between the loss and the specific parameter, and the number of paths grows exponentially with the **depth**

(number of layers) and **width** (number of nodes per layer) of the network, this quickly becomes too computationally expensive in larger networks. The key observation of the backpropagation algorithm is that we can efficiently store and reuse the gradients by 'walking backwards' through the network/computational graph. We will first give a conceptual algorithm, and then highlight some details.

Backpropagation algorithm

```
grad = $\nabla_{\hat{y}}\mathcal{L}$                                    differentiate the loss w.r.t. the network prediction
for $d$ in $D, D-1, ..., 1$:
    grad ← $\nabla_{\mathbf{z}^{(d)}}\mathcal{L} = \mathbf{grad} \odot \frac{dg^{(d)}}{dz_j^{(d)}}$       propagate through non-linearity
    $\nabla_{\mathbf{b}^{(d)}}\mathcal{L} = \mathbf{grad}$                      gradients for biases in layer $d$
    $\nabla_{\mathbf{W}^{(d)}}\mathcal{L} = \mathbf{grad} \cdot \mathbf{h}^{(d-1)}$              gradients for weights in layer $d$
    grad ← $\nabla_{\mathbf{h}^{(d-1)}}\mathcal{L} = \mathbf{grad} \cdot \mathbf{W}^{(d)}$   propagate gradients to hidden units of next layer $d-1$
```

We quickly walk through the algorithm in words. We first differentiate the loss with respect to the network output $\hat{y}$ (or equivalently some probability distribution parameters in case of maximum likelihood estimators). When then start to loop backwards through all the layers (the network has D layers), accumulating and storing the gradients. We first propagate through the non-linearities to $\mathbf{z}^{(d)}$ (this is usually an element-wise operation). Then, we compute the derivatives of the linear part $\mathbf{z}^{(d)} = \mathbf{W}^{(d)}\mathbf{h}^{(d-1)} + \mathbf{b}^{(d)}$, and store the gradients with respect to $\mathbf{W}^{(d)}$ and $\mathbf{b}^{(d)}$. Then, we propagate the gradient to the next layer by differentiating with respect to $\mathbf{h}^{(d-1)}$, and repeat. Overall, this algorithm scales linearly in the number of parameters and network depth.

6 Summary: Training neural networks

This concludes our treatment of neural network training. The only topic we did not discuss is **network initialization**. There is much more to be said about this, but the important thing to remember is that *we should not initialize all network weights to 0 or to the same value*, because this will create a symmetry that will prohibit the weights to ever become dissimilar. Therefore, we should always randomly initialize the network weights from some underlying noise distribution. In summary, we then repeatedly run the following loop:

1. **Draw** a minibatch
2. **Forward propagate**: $\hat{y}_i = f_\theta(\mathbf{x}_i)$ (or params of prob. distribution)
3. **Compute loss**: $\mathcal{L}(\hat{y}_i, y_i)$
4. **Backpropagate** the loss through the network: $\nabla_\theta\mathcal{L}$
5. **Update** the network weights with gradient descent: $\theta \leftarrow \theta - \alpha\nabla_\theta\mathcal{L}$