Deep Learning (2IMM10) - NN Primer

Vlado Menkovski
Spring 2020

The learning outcomes of this chapter:

- You are able to design and train a multi-layer percepton model

The main theme of this course is developing representations of high dimensional data. In the introduction chapter we discussed the motivation and utility of having such representations, in this chapter we will go over the underlying methods for developing such representations.

1 The Artificial Neuron

The main component of the artificial neural network is the artificial neuron. You can see a graphical representation in Figure 1. To understand what an artificial neuron is, let us look at an example of a neuron taking five inputs.

- The inputs to the neuron are given by x_0 to x_4, denoted as an input vector $\mathbf{x}$
- Edges between nodes have parameters $\mathbf{w}$, called weights, and a bias term b. The weights are used to calculate a weighted sum of the inputs, which is then offset by the bias. The weights and bias together are the parameters, denoted as θ, of the neuron.
- Finally the neuron applies some — usually non-linear — 'activation' function to the weighted sum.
- Putting everything together, this means a single neuron implements computes its output $o_\theta(x)$ as:

$$o_\theta(x) = \phi(\sum_i w_i x_i + b) = \phi(\mathbf{w}^T\mathbf{x} + b).^1$$

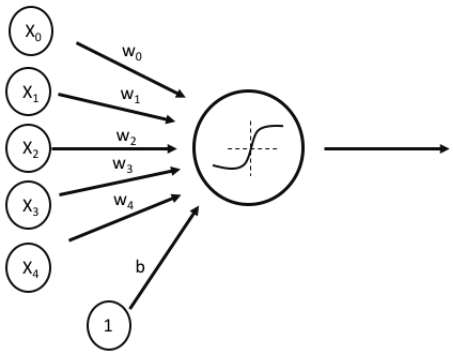


Figure 1: Artificial neuron

[1] In order to make notation a bit more concise, we will often omit writing the bias, giving to the notation $\phi(\mathbf{w}^\top\mathbf{x})$. The addition of the bias is implied. This convention is also quite common in the literature.

For example, if we have an artificial neuron taking three input variables, and if the input to the artificial neuron is a vector $\mathbf{x} = [1, 2, 3]^\top$ and the values of the parameters are $\mathbf{w} = [1, 0.5, 1]^\top$ and $b = 0.5$, the artificial neuron give the following output:

$$\begin{aligned} o_\theta(\mathbf{x}) &= \phi(\mathbf{w}^\top \mathbf{x} + b) \\ &= \phi(1 \cdot 1 + 2 \cdot 0.5 + 3 \cdot 1 + 0.5) \\ &= \phi(1 + 1 + 3 + 0.5) \\ &= \phi(5.5) \end{aligned}$$

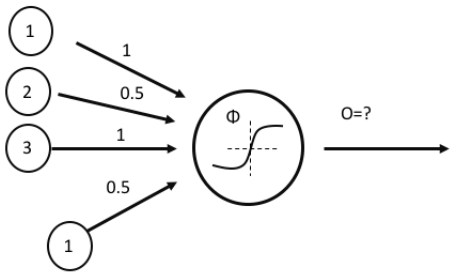


Figure 2: Artificial neuron activations

where ϕ is the activation function of the artificial neuron. The activation function can be any differentiable[2] function such as the hyperbolic tangent ($\phi(x) = \tanh(x)$) or simply a linear activation ($\phi(x) = x$).

If in our simple example we choose as an activation function $\phi(x) = x$, then for the given input, $\mathbf{x} = [1, 2, 3]^\top$, the artificial neuron will produce the value 5.5 — see Figure 2 for a schematic overview.

1.1 Linear Regression

If we choose the activation to be linear, $\phi(x) = x$, our artificial neuron becomes a linear regression model:

$$o_\theta(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$$

Let us look at how we can use this linear neuron to model a data set. As an example we will use the dataset from Figure 3.

Empirical Risk minimization

To use our neuron to model a dataset, we want to find the "best" values for our parameters $\mathbf{w}$ and b. Our dataset consists of pairs $(\mathbf{x}^{(i)}, y^{(i)})$ where $\mathbf{x}^{(i)}$ is an input vector, and $y^{(i)}$ is the corresponding value we wish to predict. Given any values for our parameters $\theta = (\mathbf{w}, b)$, our model makes predictions $\hat{y}^{(i)} = o_\theta(\mathbf{x}^{(i)}) = \mathbf{w}^\top \mathbf{x}^{(i)} + b$ for the inputs $\mathbf{x}^{(i)}$. In order to determine how good our parameters are, we can define a "loss function", $L\left(y^{(i)}, \hat{y}^{(i)}\right)$, that measures how far

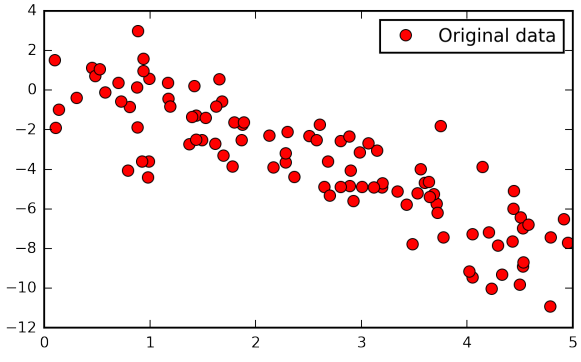


Figure 3: A dataset for which we might want to use a linear regression model.

[2]In practice it is also possible to use activation functions that are almost everywhere differentiable, such as the "Rectified Linear Unit", ReLU: $x \mapsto \max(x, 0)$.

our predictions are off, and see what the average
loss on the data set is:

$$\frac{1}{N}\sum_{i=0}^{N-1} L\left(y^{(i)}, \hat{y}^{(i)}\right).$$

This averaged loss is called the *empirical risk*. We can then define the "best" values for our parameters to be those for which the empirical risk is minimal, and "training" the model then becomes the optimization task of minimizing this empirical risk:

$$\begin{aligned}\theta_{\text{best}} &= \arg\min_{\theta} \frac{1}{N}\sum_{i=0}^{N-1} L\left(y^{(i)}, \hat{y}^{(i)}\right) \\ &= \arg\min_{\theta} \frac{1}{N}\sum_{i=0}^{N-1} L\left(o_\theta(\mathbf{x}^{(i)}), y^{(i)}\right). \end{aligned} \tag{1}$$

This process is called *Empirical Risk minimization*. The most common loss function for linear regression models is the "mean squared error" which in our case comes down to taking

$$L(y, \hat{y}) = (y - \hat{y})^2.$$

2 Gradient Descent & Backpropagation

The minimization in eq. (1) is something that we actually need to implement with an algorithm. There are many more efficient implementations of the optimization of linear regression models, however for our purposes we will study gradient descent as it allows us to scale this to the much more complex and non-linear models that we aim for in this course.

The idea behind gradient descent as an optimization algorithm is quite simple. Suppose we have a continuously differentiable function $f : \mathbb{R}^2 \to \mathbb{R}$ of two variables, whose graph looks like some hilly landscape. We start out on some hill side, and want to get to a (local[3]) minimum. What we could do is see what direction the hill goes down most steeply in, and set a step in that direction. After that step we again look at what direction the hill is steepest in and take a step down, and so on and so on. This is gradient descent: the gradient of f, $\nabla f(u, v) = \left(\frac{\partial}{\partial u} f(u, v), \frac{\partial}{\partial v} f(u, v)\right)$, at a point $(u, v) \in \mathbb{R}^2$ points in the direction in which f has the steepest increase, and the negative of the gradient of f points in the direction of steepest descent.

To apply gradient descent to our machine learning problems, we view the empirical risk, or average loss function, as a function of the model parameters. We let the model make predictions for the dataset, calculate the loss and calculate the derivatives of that loss with respect to the model parameters. Then we update the parameters. In the following subsections, we will look at gradient descent in detail.

[3]If the function that we want to minimize is a convex function with Lipschitz gradient, and if we make the right choices for our step size, gradient descent can be proven to converge to a global minimum. Our linear regression model with mean square error satisfies these conditions. However in general in machine learning the best we can hope for is a useful local minimum.

The Gradient Descent Algorithm

- Given a set of n examples in a data set $D : \{(\mathbf{x}, y)\}$.
- GD Update rule:
  - repeat until convergence

$$\begin{aligned} w &\leftarrow w - \alpha \frac{\partial}{\partial w} L(\mathbf{x}, y; \mathbf{w}, b) \\ b &\leftarrow b - \alpha \frac{\partial}{\partial b} L(\mathbf{x}, y; \mathbf{w}, b) \end{aligned} \tag{2}$$

where α is a parameter referred to as the learning rate

2.1 Gradient Descent Optimization

Gradient descent (GD) optimization is a first-order iterative optimization algorithm. The algorithm updates the parameters of the model in an iterative fashion until it converges, i.e. until the loss value does not decrease any more. The update values of the parameters are given by eq. (2). So, to get the new values of the parameters we subtract from the old values a value proportional to the gradient of the loss with respect to the parameters.

Intuitively, the gradient of the loss with respect to the parameters of the model indicates how the loss value changes by changing the parameter values. If the loss increases with an increase of a parameter, the gradient will have a positive value. As we aim at decreasing the loss we should then decrease the value of that parameter, hence the minus sign in the expression and the 'descent' in the name of the algorithm.

Because the gradient indicates how much every parameter influences the loss function, we take the amount of change to the parameters to be proportional to the gradient, multiplied by a "learning rate". The learning rate parameter α allows us to scale the step size . The value of the learning rate has a strong impact of the efficiency of the training. Think about how this happens:

- How would a large learning rate affect the training?
- How would a small learning rate affect the training?
- Does gradient descent optimization guarantee finding the parameters that minimize the loss?

To be able to implement GD optimization we need to be able to compute the gradient of the loss with respect to the parameters. As the loss is computed based on the output of our model, all components of our model need to be differentiable with respect to the parameters. We will come back to this requirement as we increase the complexity of the neural network models and include different components.

In general when developing these models we follow the following steps:

Requirements:

- Define the model and its parameters:
  - $o_\theta = \mathbf{w}^\top x + b$
  - $\theta : \{\mathbf{w}, b\}$
- Define the Loss function:
  - $L(\mathbf{x}, y; \mathbf{w}, b) = \frac{1}{N} \sum_{i=0}^{n-1} (o_\theta - y)^2$
- Specify the gradient of L with respect to the parameters $\mathbf{w}$ and b:
  - $\frac{\partial}{\partial w} L(.)$
  - $\frac{\partial}{\partial b} L(.)$

As the complexity of the models increases we will use tools such as Tensorflow to automate some of these steps and their calculations. However, to understand how this works, we will now look at how the gradients are computed.

2.2 Computing the gradient of the loss

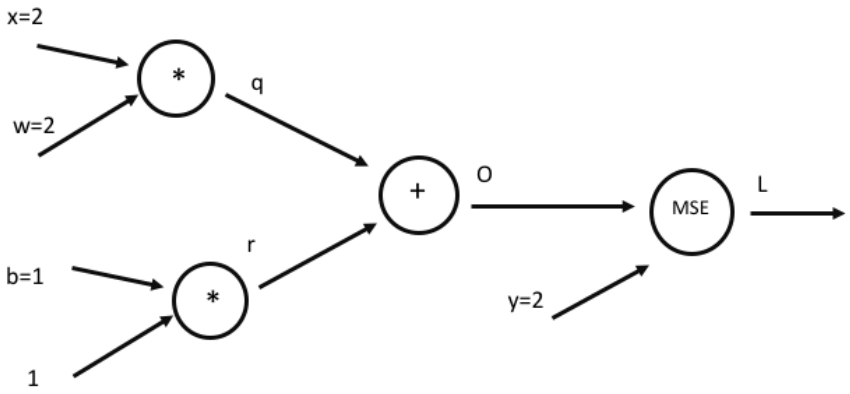


Figure 4: *Computation graph* - For a single datapoint $\{x = 2, y = 2\}$

Let us compute the gradient for our artificial neuron given a single datapoint. To help us follow the computation of the gradient we depict our model as a directed, acyclic graph: the *computation graph*. The computation graph has edges that carry values, labeled with variable names, and nodes that specify the operations on those values. Figure 4 shows the computation graph for our artificial neuron from Section 1.1 for a single datapoint and one-dimensional input.

Computing the gradient of the loss — The forward pass

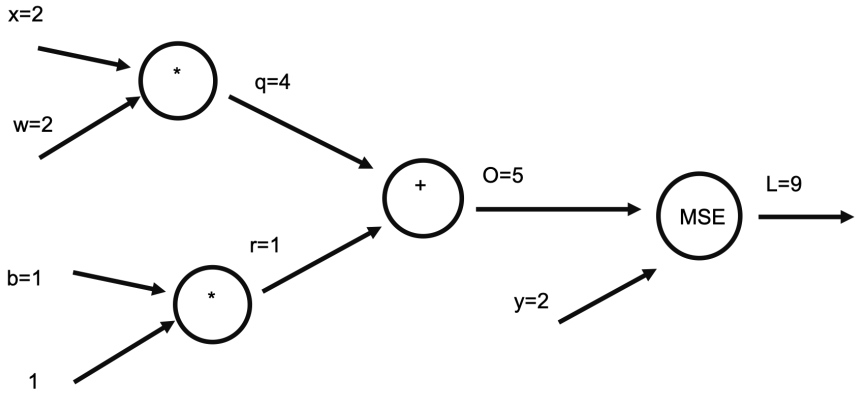


Figure 5: *Computation graph* — The forward pass

To compute the gradient of the loss with respect to to all the parameters in our model, we need to have all the intermediate values in our computation graph. We compute those in a *forward pass* over the compute graph — see Figure 5. The computations are as follows:

- $o = wx + b = 2 \cdot 2 + 1 = 5$,
- $L = \frac{1}{1}(o - y)^2 = (5 - 2)^2 = 3^2 = 9$.

Computing the gradient of the loss — The backward pass

Now that we have the intermediate values we can compute the gradient of the loss starting from the end of the computation graph and going backwards — see Figure 6.

$$
\begin{aligned}
\frac{\partial L}{\partial L} &= 1 \\
\frac{\partial L}{\partial o} &= 2(o - y) \cdot 1 = 2(o - y) \\
\frac{\partial L}{\partial q} &= \frac{\partial L}{\partial o}\frac{\partial o}{\partial q} = 2(o - y) \\
\frac{\partial L}{\partial r} &= \frac{\partial L}{\partial o}\frac{\partial o}{\partial r} = 2(o - y) \\
\frac{\partial L}{\partial w} &= \frac{\partial L}{\partial q}\frac{\partial q}{\partial w} = 2(o - y) \cdot x \\
\frac{\partial L}{\partial b} &= \frac{\partial L}{\partial r}\frac{\partial r}{\partial b} = 2(o - y) \cdot 1
\end{aligned}
$$

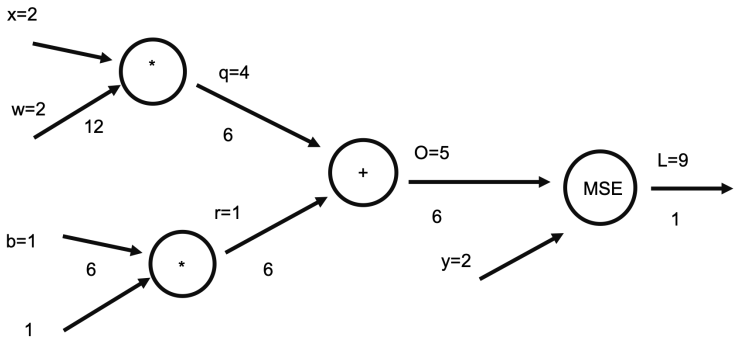


Figure 6: *Computation graph* — The backward pass

This algorithm is called backpropagation and it is basically a way to apply the derivative chain rule. The algorithm scales very well to large models. *The only requirement we have is that we need to compute the gradient of the output of any node in the compute graph with respect to its input variables.*

The Backpropagation Algorithm

- Compute forward pass
  - $o = x \cdot w$
- For each node compute the local derivatives
  - $\dfrac{\partial o}{\partial x}$
  - $\dfrac{\partial o}{\partial w}$
- Backward pass the derivative
  - apply the chain rule

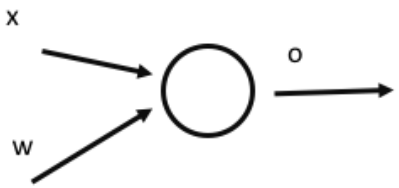


Figure 7: A node in a computation graph to perform Backpropagation on.

Computing the gradient of the loss — Updating the parameters

Now that we've computed the gradient of the loss through the backpropagation algorithm, we are ready to set a step in the gradient descent algorithm, i.e. to update the parameters. With the gradients and values from Figure 6 and learning rate $\alpha = 0.1$ this is done as:

$$\begin{aligned} w &\leftarrow w - \alpha \frac{\partial}{\partial w} L(\mathbf{x}, y; \mathbf{w}, b) \\ &= 2 - 0.1 \cdot 12, \\ b &\leftarrow b - \alpha \frac{\partial}{\partial b} L(\mathbf{x}, y; \mathbf{w}, b) \\ &= 1 - 0.1 \cdot 6. \end{aligned}$$

3 Classification

In Section 1.1 we looked at a model doing linear regression to predict continuous values, and in Section 2 we saw how we can train our linear regression model with gradient descent optimization using backpropagation. Next we aim to develop a model for classification. When doing classification, instead of predicting a continuous value for each data point, we want to predict discrete values coming from some fixed set of classes. To achieve this we specify our model's output as a discrete probability distribution over the set of target values.

$$f_c(\mathbf{x}) = P(y = c|\mathbf{x})$$

An example of binary classification

- Two input variables: x_0, x_1
- Two classes

$$c1 = \text{class 1}$$
$$c2 = \text{class 2}$$

- Output is

$$f_{c1}(\mathbf{x}) = P(y = c1|\mathbf{x})$$
$$f_{c2}(\mathbf{x}) = P(y = c2|\mathbf{x})$$

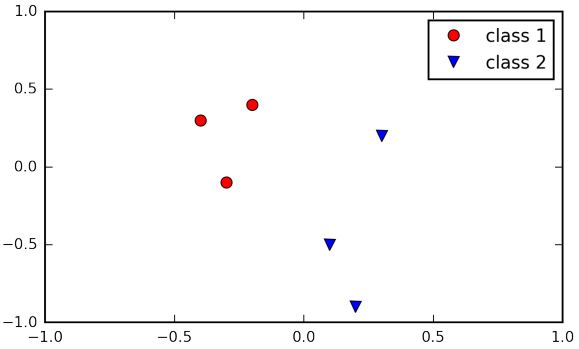


Figure 8: An example of a data set for which we might want to do classification.

To practically achieve this using the artificial neuron we use an activation function whose output has the properties of a probability distribution over a discrete set. In case of binary classification the activation typically used for this is the "logistic sigmoid" function, or "sigmoid" for short. In case of multiclass classification, where every data point belongs to only one class, the most common activation function is the "softmax"[4] function. Before we delve deeper into classification, let us first give a brief description of these two functions.

Logistic sigmoid function:

- is used for binary classification;
- is called "sigmoid" for short;
- is given by

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}};$$

- is the inverse of the "logit" function;
- can also be used when a single datapoint can belong to multiple classes;
- takes values between 0 and 1;
- its graph is shown in Figure 22e.

Softmax function:

- is used for multiclass classification;[5]
- takes a vector as its input;
- is given by

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=0}^{N} e^{x_j}};$$

[4]The softmax function is not really an activation function for a single neuron, but takes the outputs of a bunch of neurons as its input. How this is done will become clear in Section 5.2.

- its components are positive and sum to 1.

3.1 Perceptron

To understand how classification can be done using Neural Networks, we will now try to do binary classification on a dataset with two input variable — see for example the data set in Figure 8 — using a single artificial neuron with sigmoid activation.

As inputs to our artificial neuron we have the two variables, x_0 and x_1, together denoted in vector notation as $\mathbf{x}$. As with the linear regression model, we first multiply our inputs by some weights $\mathbf{w}$, then sum them and add the bias b. Finally we feed the result of this into a sigmoid activation function. A schematic overview of this is given in Figure 9. As a whole the artificial neuron implements the following:

$$\begin{aligned} o_\theta(x) &= \text{sigmoid}(w_0 x_0 + w_1 x_1 + b) \\ &= \text{sigmoid}(\mathbf{w}^\top \mathbf{x} + b) \\ &= \frac{1}{1 + e^{-(\mathbf{w}^\top \mathbf{x} + b)}}. \end{aligned}$$

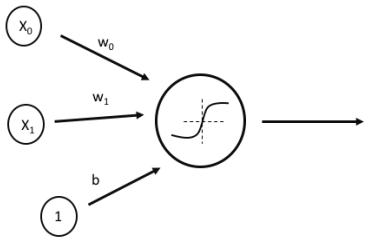


Figure 9: A schematic overview of our artificial neuron used for binary classification.

Binary classification

Now how do we use this neuron for binary classification? As mentioned earlier, we view its output as a probability distribution over our classes. More specifically, if we have classes 1 and 2, we see the output of our neuron as

$$\begin{aligned} P(y = 1 \mid \mathbf{x}) &= o_1 \\ &= o_\theta(\mathbf{x}) \\ &= \text{sigmoid}(\mathbf{w}^\top \mathbf{x} + b) \\ P(y = 0 \mid \mathbf{x}) &= o_0 \\ &= 1 - o_1. \end{aligned}$$

Then as a loss function, we use the negative log-likelihood of this Bernoulli distribution, also known as "binary cross-entropy." The likelihood of an outcome $y \in \{0, 1\}$ for a Bernoulli random variable with success probability $o_\theta(x)$ can be written as

$$P(y) = o_\theta(\mathbf{x})^y (1 - o_\theta(\mathbf{x}))^{(1-y)},$$

so that the loss becomes

$$\begin{aligned} L(\mathbf{x}, y; \theta) &= -(y \log(o_\theta(\mathbf{x})) + (1 - y) \log(1 - o_\theta(\mathbf{x}))) \\ &= -\log(o_y). \end{aligned}$$

[5] Within the field of deep learning, the softmax function is also commonly used for so-called attention mechanisms.

As with the linear regression model, we now want to use Gradient Descent to minimize the empirical risk. To be able to do this we again use back propagation on a computation graph to get the gradients. The computation graph for this model is shown in Figure 10. Try to work out the gradient of the loss with respect to the model parameters for yourself — *hint: use case distinction between* $y = 0$ *and* $y = 1$.

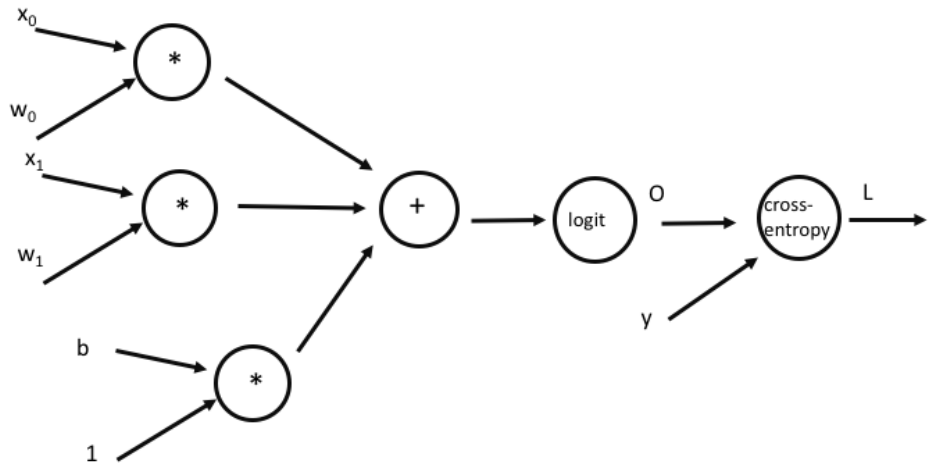


Figure 10: The computation graph for our single neuron binary classifier.

4 Multilayer Perceptron

4.1 The limitations of the Artificial Neuron

The artificial neuron model develops a linear combination of the input values to compute the output value. In the case of the classification task this linear combination forms a hyperplane boundary that separates the datapoints that are associated with a different label.

To address this limitation and enable the model to represent more complex, non-linear maps between the input and the output, we combine multiple neurons that have a non-linear activation function. First stack neurons in order to get a "hidden layer" and next stack layers on top of each other to get deeper models.

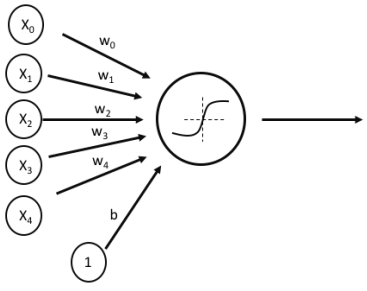


Figure 11: A single artificial neuron

Multilayer Perceptron — A hidden layer

Neurons can be stacked to form a layer of neurons. Every neuron in the layer gets the same input variables, and the outputs of the neurons together form the output vector of the layer. The neurons in the layer typically have the same non-linear[6]activation function. We can then use the output of the layer as input to an artificial neuron. Depending on the activation of the final neuron we can then use the model for non-linear regression, or for binary classification of data that is not linearly separable. This structure is an example of a "Multilayer Perceptron" (MLP), one with a single "hidden layer". A graphical representation is given in Figure 12.

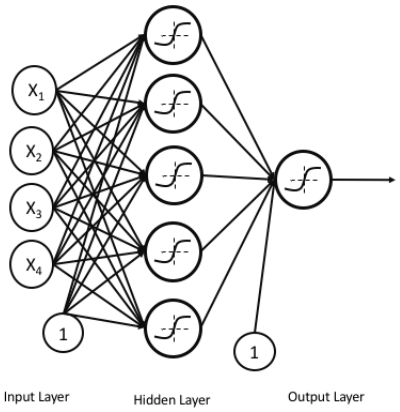


Figure 12: A Multi Layer Perceptron with a single hidden layer.

Note: This model consisting of a set of artificial neurons is an artificial neural network. The set of inputs in this model is referred to as *input layer*. The layer of neurons that we stacked on top of each other is referred to as the *hidden layer*, and the last layer that produces the output is the *output layer*.

Multilayer Perceptron — Stacking layers

In order to further increase the ability of our models to model non-linear data, we can stack layers of neurons on top of each other. The output of one hidden layer then becomes the input to the next. The number of hidden layer is called the "depth" of the network, and models with many layers stacked on top of each other are called "deep neural networks". The kind of structure that we get is shown in Figure 13.

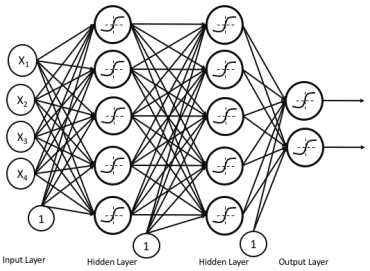


Figure 13: An MLP with multiple hidden layers

The advantage of having multiple layers is that a complex boundary between two assignments (decisions) about the input can be decomposed as a set of simpler boundaries that are then combined in the next layer. For deeper models this sequence of compositions can allow for developing highly complex maps from the input to the target variables.

Note: In principle such highly complex maps can also be achieved with a single hidden layer that has sufficiently many neurons as these neurons. However, this is significantly less efficient in terms of the number of parameters. Models that benefit from developing compositional features scale significantly better as for computing the higher level features the lower level features are re-used.

[6]Whereas with a single artificial neuron we could use linear activation in order to do regression, using a linear activation in a hidden layer does not make sense: we then have a composition of affine transformations which is itself an affine transformation, so we would not have gained any expressiveness for the model. Therefore a linear activation is typically only used in the output layer of an MLP.

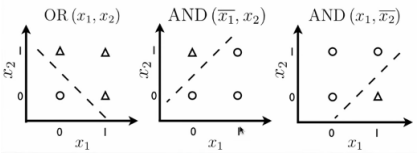


Figure 14: Dataset of binary operations that is linearly separable

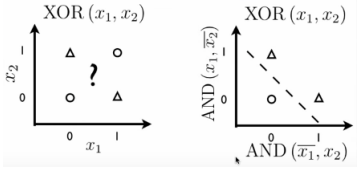


Figure 15: Data set of XOR binary operation that is not linearly separable

(The term features here refers to output values of neurons. During training neurons specialize in detecting specific pasterns of their input and can be referred to as feature detectors).

A simple example of where linearly separability is not sufficient is data generated by the binary xor operation (figs. 14 and 15).

Multilayer Perceptron — Summary so far

So far we have seen that we can stack neurons together to form layers, and that we can stack layers on top of each other to get deeper models. The models that we create this way are called Multilayer Perceptrons (MLPs). The following is a brief overview of what an MLP is:

- An MLP is a directed acyclic graph
  - where the nodes are artificial neurons
  - and the edges are parameterized connections between them.
- The nodes are organized in layers,
  - there are no connection between neurons within a layer
  - and all neurons in the same layer of the same type (have the same activation function).
- The set of inputs to the MLP is called the input layer,
- and the final layer is called the output layer.
- It is also referred to as feedforward Neural Network (meaning that there are no loops and information flows from the input directly to the output)

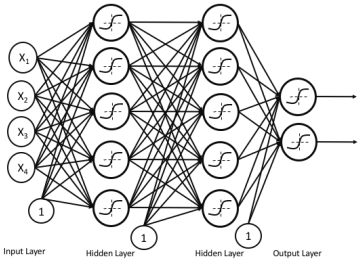


Figure 16: The structure of an MLP

We can also think of MLPs in terms of composition of functions. Each layer creates a new repre-

sentation of the input data:

$$h^{(0)} = f^{(0)}(\mathbf{x}; \theta_0)$$
$$h^{(1)} = f^{(1)}(\mathbf{h^{(0)}}; \theta_1)$$
$$y = f^{(2)}(\mathbf{h^{(1)}}; \theta_2).$$

The MLP as a whole is then a parameterised, composed, function:

$$f(x; \theta) = f^{(2)}(f^{(1)}(f^{(0)}(x; \theta_0); \theta_1); \theta_2),$$

where $\theta = (\theta_0, \theta_1, \theta_2)$. Here the $f^{(i)}$ are the layers of neurons, and the θ_i are their weights and biases.

MLP model

- Model:
  - $o_\theta = \phi_3(\mathbf{w_3}^\top \phi_1(\mathbf{w_2}^\top \phi_1(\mathbf{w_1}^\top x)))$
  - $\theta : \{\mathbf{W}\}$
- Loss function:
  - $L(\mathbf{x}, y; \mathbf{W}) = \frac{1}{n} \sum_{i=0}^{n} (o_\theta - y)^2$
- Gradient of L wrt $\mathbf{W}$:
  - $\frac{\partial}{\partial W} L(.)$

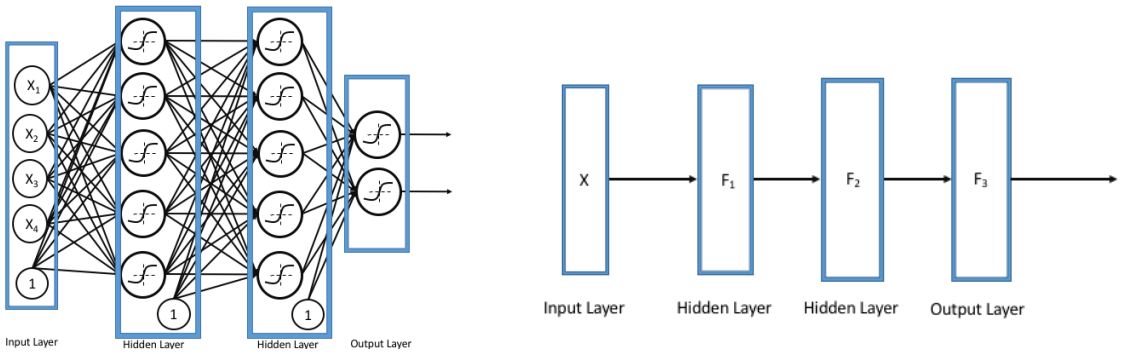
- biases omitted for brevity

Multilayer Perceptron — Layered representation

The MLP can be depicted in a more consolidated manner, where each layer is presented as a box and the connections between the layers as a single edge. This is shown in Figure 17. The advantage this gives us is that it helps us to more easily reason about complex models with many layers. Moreover, as we discuss in the following chapters we can have other kinds of layers of neurons such as convolutional layers.

Multilayer Perceptron — Backpropagation

We aim to train these models in the same way we trained the artificial neuron model: by gradient descent. In order to do this, we again compute the gradient of the empirical loss with respect to the model parameters through backpropagation. Here too it is useful to look at the model in a consolidated way: we write the computation graph in terms of the layers — see Figure 18. Now the edges carry vectors, and the nodes are operations on those vectors. Since the layers consist of artificial neurons that are not connected to each other, we can calculate the total derivative of a layer the same way as we did for a single neuron back in Section 2.2. The only difference is that now our derivative is not a vector, but a matrix: the Jacobian matrix.



(a) An MLP with boxes around the neurons constituting the layers.

(b) The final consolidated representation of an MLP.

Figure 17: Looking at an MLP as a graph of layers provides a more abstract view of the model.

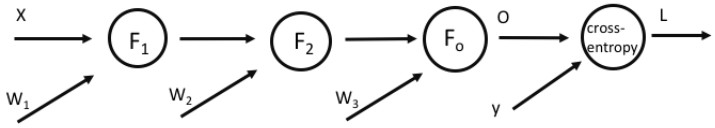


Figure 18: The computation graph in terms of layers.

Primer on the Jacobian matrix

A differentiable function $g : \mathbb{R}^n \to \mathbb{R}^m$ has at a point $\mathbf{x} \in \mathbb{R}^n$ a total derivative that is a linear map $dg_{\mathbf{x}} : \mathbb{R}^n \to \mathbb{R}^m$ that can be represented by its "Jacobian matrix"

$$J_g(\mathbf{x}) = \frac{\partial g}{\partial \mathbf{x}}(\mathbf{x}) = \begin{pmatrix} \frac{\partial g_1}{\partial x_1}(\mathbf{x}) & \cdots & \frac{\partial g_1}{\partial x_n}(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial g_m}{\partial x_1}(\mathbf{x}) & \cdots & \frac{\partial g_m}{\partial x_n}(\mathbf{x}) \end{pmatrix}.$$

These total derivatives and Jacobian matrices satisfy the following chain rule: if $g : \mathbb{R}^n \to \mathbb{R}^m$, and $h : \mathbb{R}^m \to \mathbb{R}^k$, then the total derivative of $h \circ g : \mathbb{R}^n \to \mathbb{R}^k$ and its Jacobian matrix at a point $\mathbf{x}$ are given by

$$\begin{aligned} d(h \circ g)_{\mathbf{x}} &= dh_{g(\mathbf{x})} \circ dg_{\mathbf{x}} \\ J_{h \circ g}(\mathbf{x}) &= J_h(g(\mathbf{x}))J_g(\mathbf{x}). \end{aligned}$$

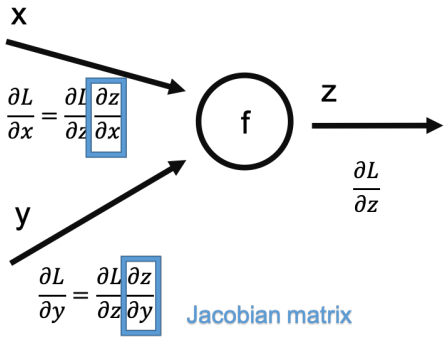


Figure 19: Derivatives of a layer.

If we now view a layer $F(\mathbf{x};\, \theta)$ with n-dimensional input as an m-dimensional vector of functions

$$F(\mathbf{x};\, \theta) = \begin{pmatrix} f_1(\mathbf{x};\, \theta_1) \\ \vdots \\ f_m(\mathbf{x};\, \theta_m) \end{pmatrix}$$

where $f_1, \ldots, f_m$ are the individual neurons with respective parameters $\theta_1, \ldots, \theta_m$ and $\theta_i = (w_{i,1}, \ldots, w_{i,n}, b_i)$, then its Jacobian matrix with respect to the parameters can be written as the following **block matrix**:

$$J_F = \frac{\partial F}{\partial \theta} = \begin{pmatrix} \frac{\partial f_1}{\partial \theta_1} & 0 & 0 & \cdots & 0 & 0 \\ 0 & \frac{\partial f_2}{\partial \theta_2} & 0 & \cdots & 0 & 0 \\ 0 & 0 & \frac{\partial f_3}{\partial \theta_3} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & \frac{\partial f_m}{\partial \theta_m} \end{pmatrix},$$

where $\frac{\partial f_i}{\partial \theta_i} = J_{f_i}$ is a $1 \times (n+1)$ matrix:

$$\frac{\partial f_i}{\partial \theta_i} = \begin{pmatrix} \frac{\partial f_i}{\partial w_{i,1}} & \cdots & \frac{\partial f_i}{\partial w_{i,n}} & \frac{\partial f_i}{\partial b_i} \end{pmatrix}.$$

Note that the off-diagonal blocks are 0-matrices because the neurons in our MLP do not share weights. This will be different when we get to convolutional layers. *Question: do we have the same for the Jaccobian matrix of the layer with repect to its input? Why or why not?*

Multilayer Perceptron — Backpropagation continued

Now lets see at what backpropagation looks like for an MLP with three hidden layers. Let L be our loss function, and let our MLP be given by $F_3 \circ F_2 \circ F_1$, where F_i is a layer with parameters θ_i (note that θ_i takes the role of θ in the primer on the Jacobian matrix above).As shown in Figure 20, we call the output of F_3 o, that of F_2 r, and that of F_1 q. We can then do back propagation as follows:

$$\begin{aligned}
\frac{\partial L}{\partial \theta_3} &= \frac{\partial L}{\partial o}\frac{\partial o}{\partial \theta_3}, \\
\frac{\partial L}{\partial r} &= \frac{\partial L}{\partial o}\frac{\partial o}{\partial r}, \\
\frac{\partial L}{\partial \theta_2} &= \frac{\partial L}{\partial r}\frac{\partial r}{\partial \theta_2}, \\
\frac{\partial L}{\partial q} &= \frac{\partial L}{\partial r}\frac{\partial r}{\partial q}, \\
\frac{\partial L}{\partial \theta_1} &= \frac{\partial L}{\partial q}\frac{\partial q}{\partial \theta_1},
\end{aligned}$$

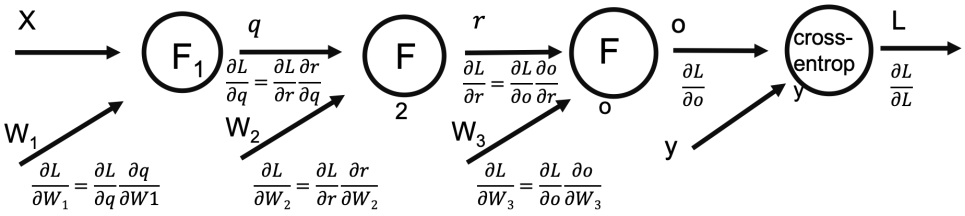


Figure 20: Backpropagation over the computation graph of an MLP with three hidden layers.

where we know how to compute the local gradients from Section 2.2 and the primer on the Jacobian matrix.

5 Model Design

So far we have seen how we can group artificial neurons together into layers, and stack layers on top of each other to get deeper neural networks. We have seen how we can build regression models and a model for binary classification, and we have seen how we can train these models by minimizing some loss function through gradient-descent. However, in the examples we have seen so far, we have brushed over some of the choices that were made. In this section we will zoom in on those choices that need to be made when developing a model. In later chapters you will learn about more techniques such as different types of layers, and stochastic regularization methods, that all come with their own design decisions, but for now we will focus on the following decisions and considerations:

- What loss function we are going to minimize during training
- What model design we are going to use (number of neurons and layers, activation function for the hidden layers)
- What activation the output layer will have
- What training parameters we will use for the gradient descent algorithm (e.g. learning rate)

To to be able to make these decisions in an informed way, we need to:

- Specify the input to our model

Figure 21: The MNIST dataset. Pixels with a higher value are depicted here as darker.

- Specify the output of our model (and how we are going to interpret it).

We will go over these decisions using classification on the MNIST dataset shown in Figure 21 as an example.

5.1 The MNIST dataset

We want to write a model that can take pictures of handwritten digits as its input and output what digit is in the picture. The dataset we are given, the MNIST dataset, consists of 28×28 gray-scale images. Each pixel has a single number associated to it indicating its brightness. These values are integers between 0 and 255. The images all belong to one of ten classes, an image belonging to class i meaning that the image contains the digit i. A selection of images from the MNIST dataset is shown in Figure 21.

If we want to write a neural network that can classify these images, a good start is to look at the data. Neural networks tend to perform better on data lying in a range of magnitude $O(1)$. Large numbers in the data make that a small difference in parameters in one layer will have a very large impact on the input the next layer receives. This is detrimental to its ability to be trained by gradient descent. On the other hand, when all data is very small, e.g. in a range of $(0, 10^{-3})$, the weights and biases of the network need to be very large to come to output of the right size, again negatively impacting the ability of the model to by trained by gradient descent. Keeping this in mind we will pre-process the data by transforming the integers to floating-point numbers, and scaling to the interval $[-1, 1]$ by the following map:

$$x \mapsto \frac{x - 127.5}{127.5}.$$

Besides that, we will reshape the 28×28 arrays to vectors of size 784 to feed to our neural network.

5.2 Output and Loss

Now that we have the input ready, let's look at the output. As we discussed in Section 3, doing classification with neural networks is typically done by outputting a probability distribution over the classes. To get matching labels, we will perform a so called *one-hot encoding*: we map an integer i to a vector e_i of length 10 where the i^{th} element is 1 and all other elements are 0, as shown in eq. (3) — here $\delta_{i,j} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$

$$i \mapsto e_i = (\delta_{i,j})_{j=0,\dots,9}. \tag{3}$$

In order to output a probability distribution over our ten classes, we will use the softmax function. To do this, we take our last layer of neurons to have ten neurons, each of which will correspond to its own class. The output $(v_0, \dots, v_9)$ of these neurons will then be fed to the softmax function

$$\text{softmax} : (v_0, \dots, v_9) \mapsto \left(\frac{e^{v_j}}{\sum_{i=0}^{9} e^{v_i}} \right)_{j=0,\dots,9}.$$

We want the output distribution for an image to give a lot of mass to the correct class, and very little to all other classes. That is, we want the likelihood of the correct answer to be as high as possible. To rephrase this as a minimization problem: we will aim to minimize the negative log-likelihood. Now if we have label y with one-hot encoding e_y, and we have a probability distribution $(p_0, \dots, p_9)$ over our classes (where all the p_i are positive, and their sum is 1), then the likelihood of y is

$$p(y) = \prod_{i=0}^{9} p_i^{e_{y,i}},$$

so the negative log-likelihood is the *categorical cross-entropy*[7]

$$\begin{aligned} L(y, p) &= -\sum_{i=0}^{9} e_{y,i} \log(p_i) \\ &= -\log(p_y). \end{aligned} \tag{4}$$

Let us take a step back and look at how we have come up with this loss function. Just like with the binary cross-entropy, we output a probability distribution, and we intend to maximize the likelihood of the true labels. This way, the loss function follows kind of naturally[8]from the fact that we are trying to predict a probability distribution, and from the type of distribution. When trying to predict continuous outcomes with a regression model, we based our loss simply on a distance between our predictions and the correct labels. We could however view this in the same light: if we view the outcome of the model not as a single value, but as the center for a normal distribution with fixed variance, the resulting average negative log-likelihood would (up to an additive constant which has no effect on training with gradient-descent) be a scaled version of the mean squared error.

In general there are many loss functions you can use to train a neural network. What loss function does work, and what doesn't depends strongly what the output of your network represents. It is

[7] Cross-entropy is really an information-theoretic measure of error between two probability distributions. The categorical cross-entropy in eq. (4) is a measure of how far we are off when using the incorrect probability distribution p instead of the true (deterministic) distribution e_y.

therefore important to consider what kind of values you are trying to predict, and how you are going to interpret the output of your neural network. Moreover, it is important to remember that if you want to train your network using gradient-descent or other gradient-based methods, the loss function needs to be differentiable. Additionally it is important to have some sort of monotonicity in that predictions that are further off target should give a higher loss to prevent getting stuck in low quality local minima. Most of the time one of the following three losses will be an appropriate choice:

- Mean Squared Error if you are dealing with a regression task;
- Binary Crossentropy if you are doing with binary classification;
- Categorical Crossentropy if you are doing multiclass classification.

Note also the role the final activation plays: with regression our final activation was linear, with binary classification it was a sigmoid function, and now with multiclass classification it is a softmax function. In general it is important to use an activation in the output layer that allows the model to give good predictions. If the values you are trying to predict lie in a small and fixed range, a linear activation will make it hard for your model to stay within that range. On the other hand, if there is no clear bound on the output — like when doing regression — a bounded activation function like a sigmoid will not allow your model to approximate the correct outcomes.

5.3 Network Design

We have pre-processed our data, we have decided upon the type of output we want our model to give, chosen an activation for the final layer, and defined a loss function. Now it is time to design the neural network itself. How many layers should we use? How many neurons should each layer have? And what activations are we going to use?

A large part of this is simply trial and error. Adding or removing a single layer can have large impacts on model performance, as can adding or removing neurons to/from layers. There are a couple of things however that you can keep in mind to help you make informed guesses.

Vanishing or Exploding Gradients

One of the main problems that can make training neural networks difficult is the gradients getting either too large (exploding) or too small(vanishing). This is because, as we saw in section 4, composition of layers results in a product of derivatives. If the derivatives are all smaller than 1, the product quickly vanishes to 0, but if they are all larger than 1 the product grows exponentially.

Activation Functions

Originally the go-to activation function for hidden layers were sigmoidal functions such as the logistic sigmoid and the hyperbolic tangent. These functions however suffered greatly from the

[8]The logarithm is really used here for numerical stability. Throughout this course you will see the theory of probability distributions being use in combination with neural networks time and again, and often the logarithm will play a role. This is not only because of that numerical stability, but also because it simplifies various expressions, and because many information-theoretic concepts related to probability distributions are defined using the logarithm.

vanishing gradient problem, making it hard to train deep networks with sigmoidal activations in the hidden layers. This shows that when choosing an activation function it is good to keep the vanishing/exploding gradient problem in mind. One approach to prevent this, is to take an activation function that is the identity on the positive half of its domain. The best known example of this is the Rectified Linear Unit (or ReLU) given by

$$\text{ReLU}: \ x \mapsto \max(0, x)$$

and depicted in Figure 22a. Because the derivative of the ReLU function on the positive half line is precisely 1, this helps against vanishing or exploding gradients. As a consequence, the advent of the ReLU allowed for much faster and better training of neural networks than before. However, the ReLU comes with its own problems when it comes to gradients: since it is constant for negative input, a neuron giving negative output to its ReLU will have a zero gradient from which it can never recover through gradient descent. This is called the "dying ReLU" problem and the neuron in question is said to be dead.

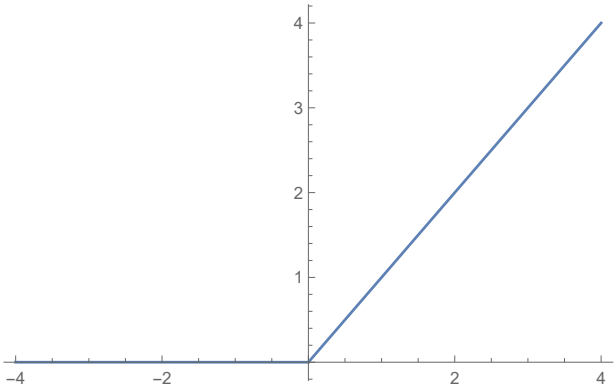
This dying ReLU problem lead to the introduction of a number of other activation functions such as the

- Leaky ReLU: $x \mapsto \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$
  - hyper parameter $\alpha \in (0, 1)$
  - called "parameterized ReLU" when α is learned through gradient-descent
- Exponential Linear Unit (ELU): $x \mapsto \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$
  - has several variations such as the Parameterized Exponential Linear Unit, and the Scaled Exponential Linear Unit.
- Softplus: $x \mapsto \log(1 + e^x)$
  - is a smooth approximation to the ReLU
  - its derivative is the logistic sigmoid which can be thought of as a smooth approximation to the Heaviside step function which is the derivative of the ReLU.

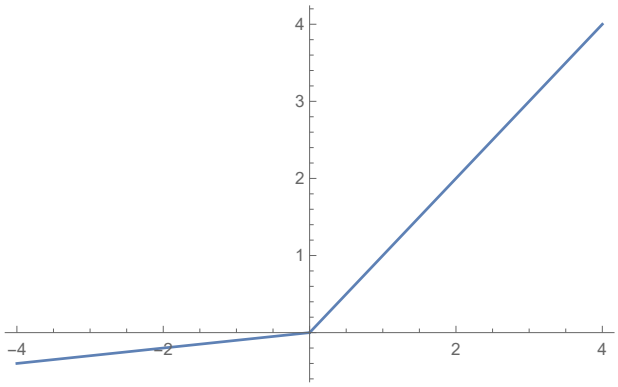
These all solve the dying ReLU problem, but for some of these the downside is that their gradients are more costly to compute.

Depth and Width of the Network

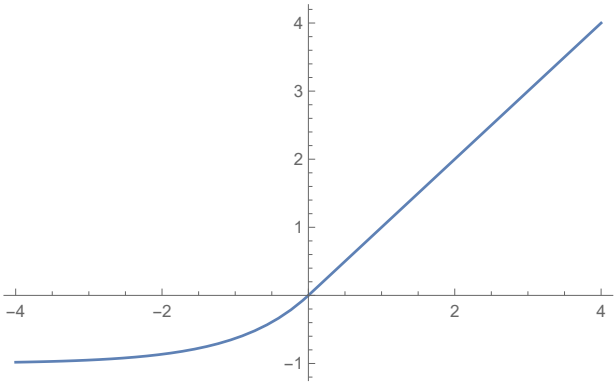
When you hear about neural networks it often seems that bigger is better. The state of the art on many tasks for many data sets consists of huge neural networks with millions of neurons arranged in a large number of layers. These models have managed to develop highly efficient representations of the data that in turn have allowed for breakthrough performance on many tasks. Does this mean, however, that we should also make our classifier on the MNIST set dozens of layers deep? The answer to that question surprisingly is a resounding *no* for several reasons.



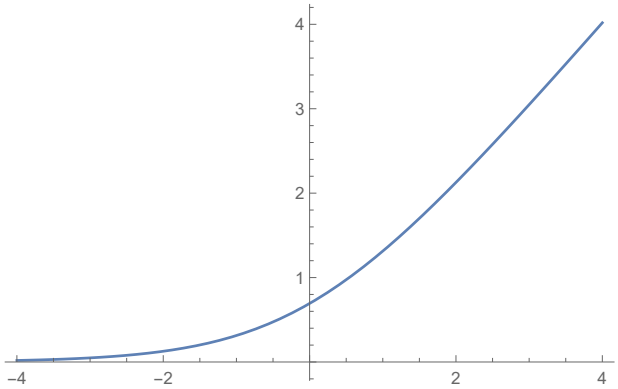
(a) The graph of a Rectified Linear Unit



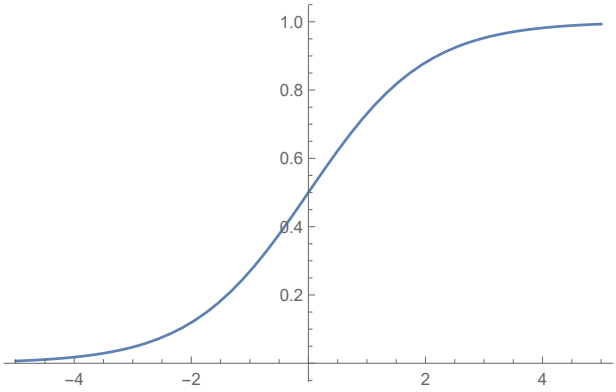
(b) The graph of Leaky-ReLU.



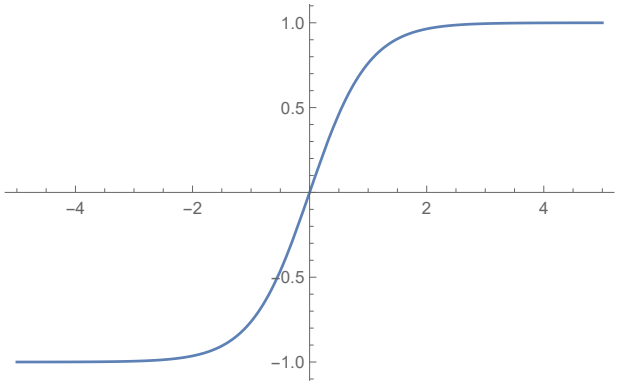
(c) The graph of an Exponential Linear Unit.



(d) The graph of the softplus function.



(e) The graph of the logistic sigmoid function.



(f) The graph of the hyperbolic tangent function.

Figure 22: The graphs of various commonly used activation functions.

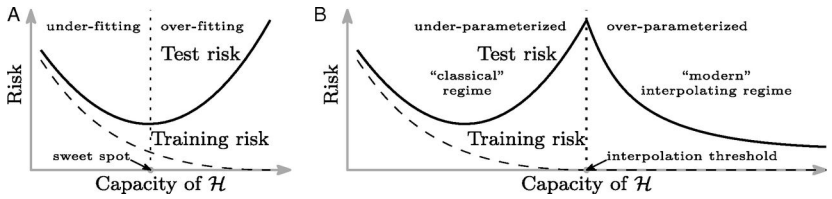


Figure 23: Bias-variance in overparametarize models
9

The first and probably most important reason lies in the layers we are using: the large neural networks you hear about typically combine different kinds of layers, such as convolutional layers, that allow for very deep models, and combine them with a myriad of techniques that help in training neural networks. Later in this course we will learn about some of these layers and techniques, but for now we are using only so-called "fully connected" layers where the neurons do not share any weights, and where the neurons within a layer have no connections to each other. These layers don't really allow for very deep networks and as mentions as recently as in 2017 many of the best performing networks of this kind were only four layers deep. Several techniques can help with training deeper networks but it is good to keep in mind that especially with fully connected networks more is not always better.

Another reason why more is not necessarily better is resources. Especially in image processing the state of the art consists of huge (convolutional) neural networks that have been trained over long periods of time on very expensive hardware. However, when you want to solve an actual problem using neural networks, you might not have similar resources available. This means you will likely have to make a trade-off between accuracy of the model and the cost of training (and even using) it.

Finally a problem that often occurs when using very powerful (deep) models, especially to predict relatively simple data, is that of *overfitting*. This is when a network is making predictions based on the peculiarities of the specific data it is trained on instead of on general patterns. The model might then perform extremely well on the data it is trained on, but will give much lower quality predictions on new data. This is a common problem in deep learning, and there is a large number of techniques that try to mitigate it, but one of the most effective (albeit rigorous) strategies is to simply reduce model complexity. Often that comes down to reducing the number of neurons in the model. Most well recognized deep neural network models are, however, overparameterized (have larger number of parameters than needed to capture the complexity of the map). The effects of the overparameterization of models in machine learning is an active field of research. Recent work indictes that the usual bias-variance trade-off has significantly different behaviour in these modelsfig. 23.

So, when designing deep neural networks we do not consider necessarily only the complexity of the map that the model needs to achieve, but also how the number of parameters affects the optimization process. In many cases an overparameterized model allows for the SGD optimization to be much more efficient in finding parameter values that yield good generalization properties. This does not mean that these models do not overfit, but on the contrary that controlling the overfitting with well designed regularisation techniques is a key component of the design.

[8]Here too the vanishing and exploding gradient problems play a large role: the more layers you have, the more factors there are in the overall derivative, thus the more opportunity it has to vanish or explode.

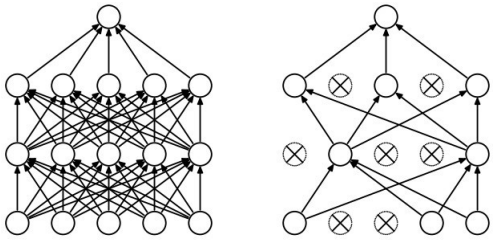


Figure 24: Dropout regularization

5.4 Regularization

Regularization techniques are one of the key breakthroughs that allowed for the success of deep learning. They both allow for significantly larger number of parameters as well as training the model longer. Both of these aspects of developing deep neural network models are key to achieve the performance that they have on high dimensional data.

Regularization can be applied to both the values of the weights as well as the values of the activation in neural networks.

Well known techniques for regularization in machine learning such as L_1 and L_2 regularization also apply in neural networks. The L_1 (LASSO) results in sparse values. This type of regularization may be particularly desirable on the activations of the neurons if we would like to have a representation of our data that is sparse. We revisit this idea when we discuss sparse autoencoders in the following chapters. On the other hand L_2 (ridge) regularization will induce lower values. When applied to the weights, it limits the capacity of the model and hence it can control for overfitting.

One of the most successful regularization techniques in deep learning is dropoutfig. 24. Dropout is typically applied to the activations of the neurons. Dropout is applied during training of the model by randomly setting a set fractions of the activations of neurons in a particular layer to zero. The idea behind dropout is to prevent the neurons in one layer to specifically depend (or co-adapt) to the activations of some of the neurons in the previous layer. In general, we would like to avoid store the datapoints in a form of memory and then recall them during inference, but rather to form an efficient representation of the data that will allow for generalization on the test set. When neurons are prevented from co-adapting to each other[10], unique pathways of activation to specific training datapoints are disabled. In contrast the neural network, starts to develop a distributed representation of the data. The distributed representation[11] allows for compositionality of the representation. Features from the lower layers can be composed to form high level features in the subsequent layers that results in highly efficient representation.

Note: Many other commonly used regularization methods are ommitted due to the space in these lecture notes. The "Deep Learning" book (Ian Goodfellow and Yoshua Bengio and Aaron Courville (2016), chapter 7 is on regularzation. It is work to further highlight a the more recent method, the BatchNormalization[12] that has enabled significant success on many modern neural networks.

[10]Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." The journal of machine learning research 15.1 (2014): 1929-1958.

[11]Hinton, Geoffrey E. "Distributed representations." (1984).

[12]Ioffe, Sergey, and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." International Conference on Machine Learning. 2015.

This method particularly improves the training process, by controlling how the distribution of the activations of the layer changes as they are exposed to new data.

5.5 Initialization

The initialization of the parameters of the neural networks are also an important aspect of the model design. As the training of these models is a stochastic process, the initial values of the parameters can result in significantly different outcomes. This is also an active area of research and out of the scope of this document.

Note: For further reading on initialization please refer to Grolot[13] and He[14].

6 Training

The training process of neural networks is stochastic and comes with its one set of configuration parameters and design decisions. One of the most salient parameters is already in the update step of gradient descent, the learning rate eq. (5).

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\mathbf{x}; \theta) \tag{5}$$

Recent methods actually adapt the learning rate parameter rather than keeping it at a constant rate. Specifically, with the introduction of the momentum technique eq. (6). Momentum allows the model to take updates with larger steps if we have been going in the same direction for a while, and smaller steps when we are changing directions, which corresponds to the notion of momentum in mechanics.

$$\begin{aligned} v &\leftarrow \gamma v - \alpha \nabla_\theta L(\mathbf{x}; \theta) \\ \theta &\leftarrow \theta - v \end{aligned} \tag{6}$$

Without going into further details, some of the commonly used variations of the SGD algorithm that use such techniques are: Nestorov momentum, AdaGrad, AdaDelta Adam and RMSProp.

One other key parameter for which a decision has to be made is the batch size. The subsampling of the dataset that SGD does, introduces a type of noise in the updates that helps avoiding local minima during training. The more we subsample, or the smaller the batch size, the larger the amount of noise. So, choosing the right batch size has significant influence on the training. One other consideration is that for many high dimensional cases and large model the computational resources needed can actually limit the batch size. To implement the update step, we have to hold in memory the activations of the forward pass to be able to compute the backward pass. We do this typically in parallel for the whole batch. Using large batches can then lead to out of memory conditions.

[13] Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." Proceedings of the thirteenth international conference on artificial intelligence and statistics. 2010.

[14] He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." Proceedings of the IEEE international conference on computer vision. 2015.