Advanced Machine Learning (Semester 1 2023)

Training and Tuning of Neural Networks

Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

May 8, 2023

Recap

Using neural networks requires an **understanding of their characteristics**:

- Choice of model: Depending on the data representation and the application. Overly complex models are slow learning.

Recap

Using neural networks requires an **understanding of their characteristics**:

- Choice of model: Depending on the data representation and the application. Overly complex models are slow learning.
- Learning algorithm: Trade-offs exist between learning algorithms. Many algorithms will work well with the correct hyperparameters for training on a **particular** data set. However, selecting and tuning an algorithm for training on unseen data requires experimentation.

Recap

Using neural networks requires an **understanding of their characteristics**:

- Choice of model: Depending on the data representation and the application. Overly complex models are slow learning.
- Learning algorithm: Trade-offs exist between learning algorithms. Many algorithms will work well with the correct hyperparameters for training on a **particular** data set. However, selecting and tuning an algorithm for training on unseen data requires experimentation.
- Robustness: If the model, cost function and learning algorithm are selected **appropriately**, the neural network can become robust.

Recap

Neural Networks need to be **trained** by adjusting the weights:

Backpropagation: new weights are calculated according to minimizing a loss function which compares actual output to expected training output

Recap

Neural Networks need to be **trained** by adjusting the weights:

Backpropagation: new weights are calculated according to minimizing a loss function which compares actual output to expected training output

Loss function: a function of the difference between expected outputs $\hat{y}$ vs. training outputs y

Recap

Neural Networks need to be **trained** by adjusting the weights:

Backpropagation: new weights are calculated according to minimizing a loss function which compares actual output to expected training output

Loss function: a function of the difference between expected outputs $\hat{y}$ vs. training outputs y

Gradient descent: calculates derivatives and by doing so optimizes the loss function

Recap

General rules for training of machine learning algorithms:

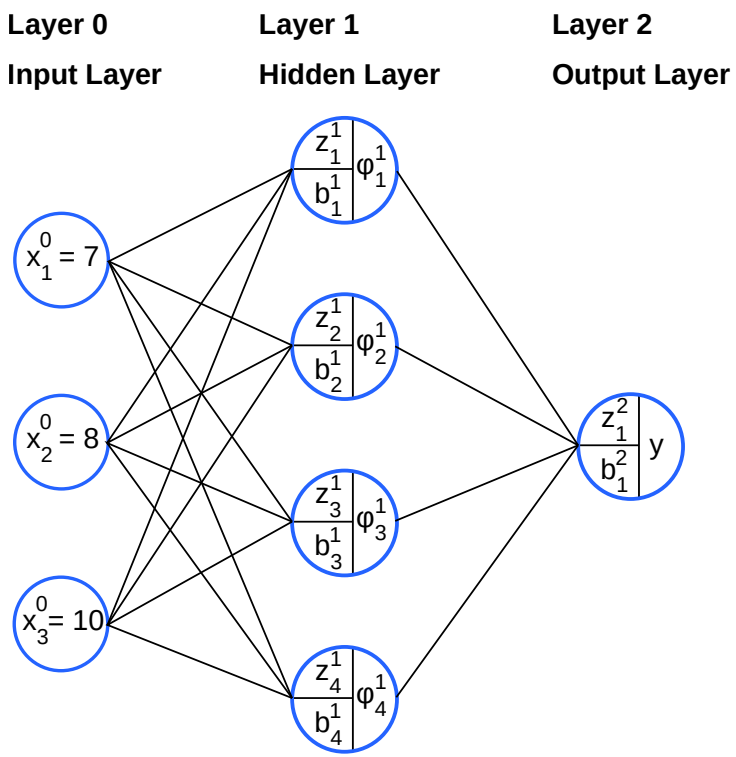
separate data used for training: the initial **modeling set** is split into training, validation and holdout data sets

- **training data**: fit parameters of the network
- **validation (test) data**: evaluate the performance on unseen data
- **holdout data**: assess the final preformance of the tuned model

Recap: Backpropagation in a Multilayer NN

example: 3-4-1 NN, objective: binary classification

dataset: three features; target class: 1 or 0, $\hat{y} = 1$



Layer 0

	i	value
x_i^0	1	7
	2	8
	3	10

Layer 2

	i	j=1
w_{ji}^2	1	0.088
	2	0.171
	3	0.005
	4	-0.04
b_j^2		0

Layer 1

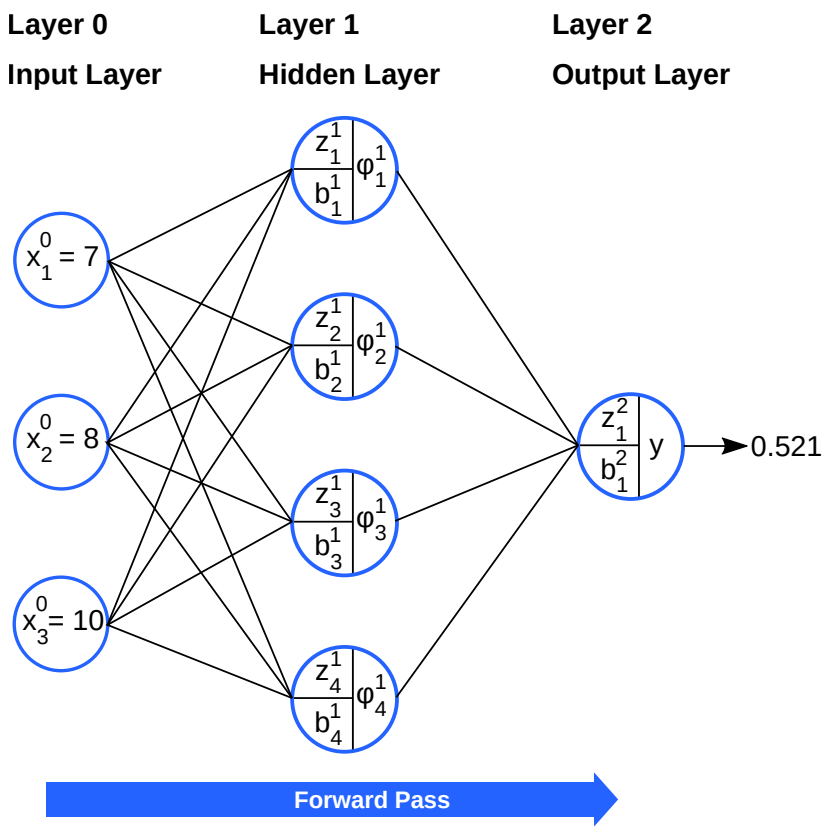
	i \ j	1	2	3	4
w_{ji}^1	1	0.179	-0.186	-0.008	-0.048
	2	0.044	-0.028	-0.063	-0.131
	3	0.01	-0.035	-0.004	0.088
b_j^1		0	0	0	0
φ_j^1		0.845	0.132	0.354	0.377

i: node in the previous layer
j: node in the current layer

Recap: Backpropagation in a Multilayer NN

example: 3-4-1 NN, objective: binary classification

dataset: three features; target class: 1 or 0, $\hat{y} = 1$



Layer 0

	i	value
x_i^0	1	7
	2	8
	3	10

Layer 2

	i	j=1
w_{ji}^2	1	0.088
	2	0.171
	3	0.005
	4	-0.04
b_j^2		0

Layer 1

	i \ j	1	2	3	4
w_{ji}^1	1	0.179	-0.186	-0.008	-0.048
	2	0.044	-0.028	-0.063	-0.131
	3	0.01	-0.035	-0.004	0.088
b_j^1		0	0	0	0
φ_j^1		0.845	0.132	0.354	0.377

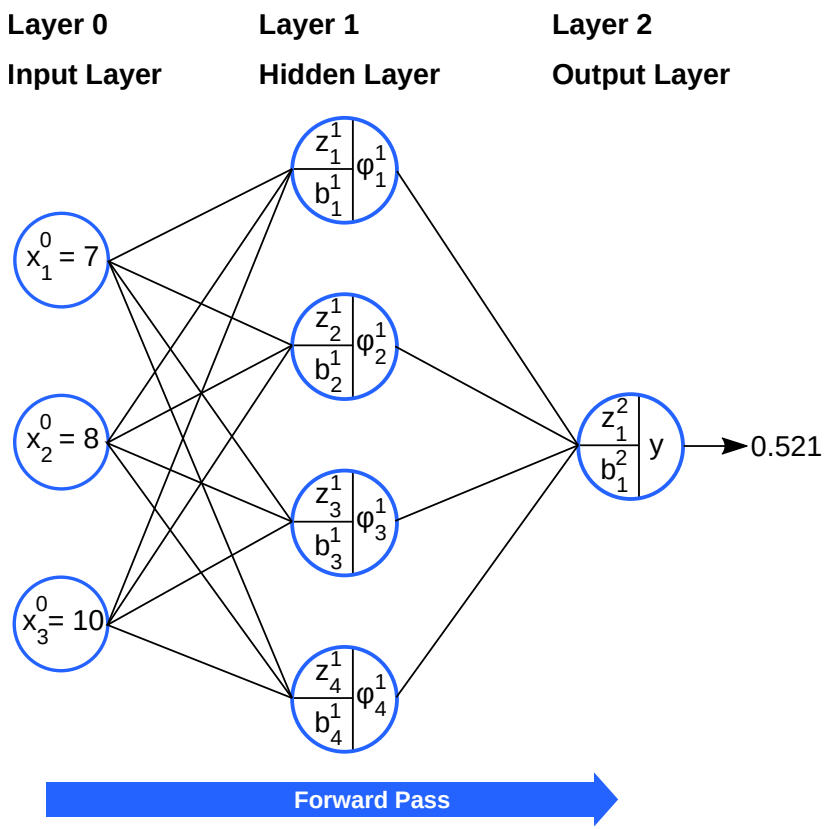
i: node in the previous layer
j: node in the current layer

A forward pass yields a prediction $y = 0.521$ of the target $\hat{y} = 1$.

Recap: Backpropagation in a Multilayer NN

example: 3-4-1 NN, objective: binary classification

dataset: three features; target class: 1 or 0, $\hat{y} = 1$



Layer 0

	i	value
x_i^0	1	7
	2	8
	3	10

Layer 2

	i	j=1
w_{ji}^2	1	0.088
	2	0.171
	3	0.005
	4	-0.04
b_j^2		0

Layer 1

	i \ j	1	2	3	4
w_{ji}^1	1	0.179	-0.186	-0.008	-0.048
	2	0.044	-0.028	-0.063	-0.131
	3	0.01	-0.035	-0.004	0.088
b_j^1		0	0	0	0
φ_j^1		0.845	0.132	0.354	0.377

i: node in the previous layer
j: node in the current layer

Loss function E defined for m training samples: $E(\theta) = \frac{1}{m} \sum_{i=1}^{m} E_i(y, \hat{y})$

here: $m = 1$, so E reduces to $E(\theta) = E(y, \hat{y})$ where θ are the parameters

Backpropagation in a Multilayer Neural Network

The Loss Function:

Here we use **Binary cross-entropy loss** which is suitable for a binary classification task*. Applying it to the forward-pass in our example to the model prediction y yields:

$$\begin{aligned} E(\theta) &= -[\hat{y}\ln(y) + (1-\hat{y})\ln(1-y)] \\ &= -[1\ln(0.521) + (1-1)\ln(1-0.521)] \\ &= 0.652 \end{aligned}$$

recap: The value of y depends only on $\theta = (w, b)$ and the input.

* We will see see more on choosing the loss function later.

Backpropagation in a Multilayer Neural Network

The parameters at each layer are updated with the following equations:

$$w_{t+1} := w_t - \eta \frac{\partial E(\theta)}{\partial w}$$
$$b_{t+1} := b_t - \eta \frac{\partial E(\theta)}{\partial b}$$

where t is the learning step, η is the learning rate which determines the rate at which the weights and biases are updated. We will use $\eta = 0.5$ (arbitrary choice).

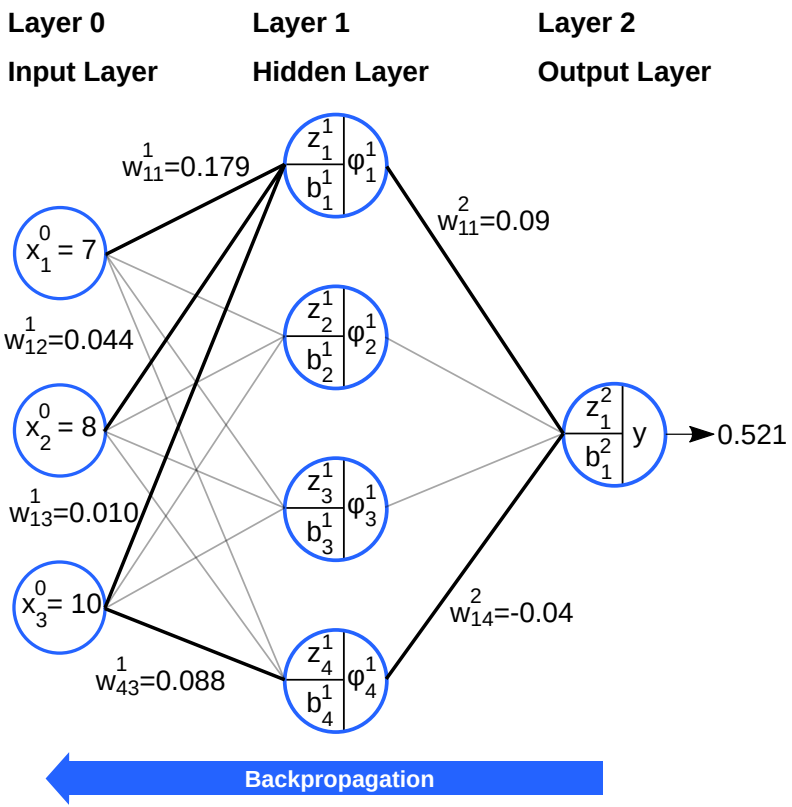
From this, the update amounts becomes

$$\Delta w_t = -\eta \frac{\partial E(\theta)}{\partial w}$$
$$\Delta b_t := -\eta \frac{\partial E(\theta)}{\partial b}$$

Backpropagation in a Multilayer Neural Network

We can now start **Updating Parameters on the Output-Hidden Layer**. Backpropagation works backwards from the output layer to layer 1.

As an example, we update w_{11}^2 and b_1^2.



We first calculate the derivatives of the **weights**.

By chain rule of differentiation, we have

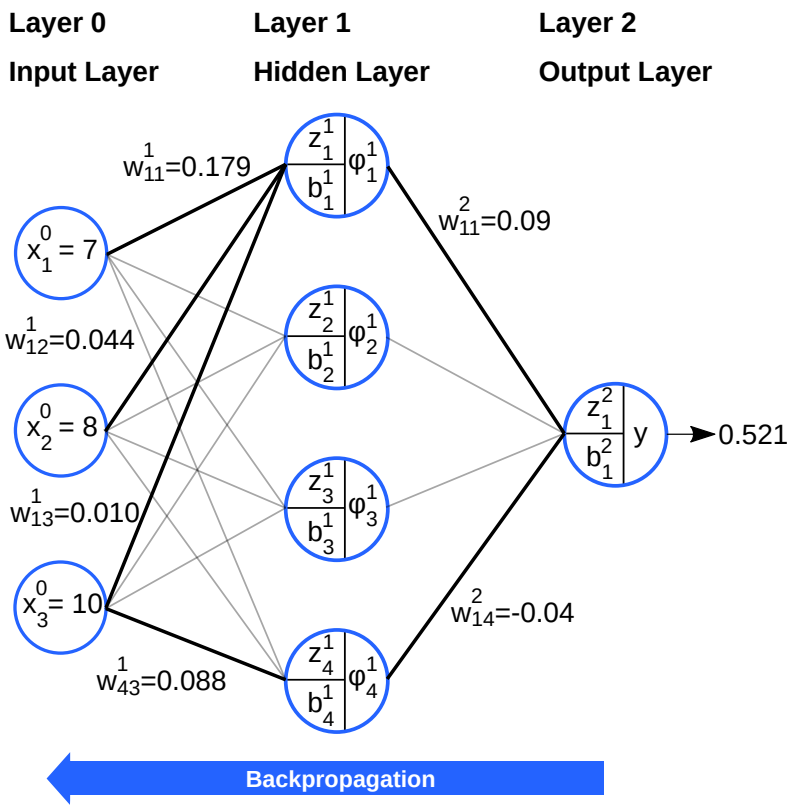
$$\frac{\partial E}{\partial w_{11}^2} = \frac{\partial E}{\partial y}\frac{\partial y}{z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2}$$

$$\begin{aligned}\frac{\partial z_1^2}{\partial w_{11}^2} &= \frac{\partial}{\partial w_{11}^2}(\varphi_1^1 w_{11}^2 + \varphi_2^1 w_{12}^2 \\ &\quad + \varphi_3^1 w_{13}^2 + \varphi_4^1 w_{14}^2 + b_1^2) \\ &= \varphi_1^1\end{aligned}$$

Backpropagation in a Multilayer Neural Network

We can now start **Updating Parameters on the Output-Hidden Layer**. Backpropagation works backwards from the output layer to layer 1.

As an example, we update w_{11}^2 and b_1^2.



We first calculate the derivatives of the **weights**.

By chain rule of differentiation, we have

$$\frac{\partial E}{\partial w_{11}^2} = \frac{\partial E}{\partial y}\frac{\partial y}{z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2}, \quad \frac{\partial z_1^2}{\partial w_{11}^2} = \varphi_1^1$$

Next is the derivative of the activation function, here a Sigmoid:

$$\varphi(z) = \frac{1}{1+\exp(-z)}$$

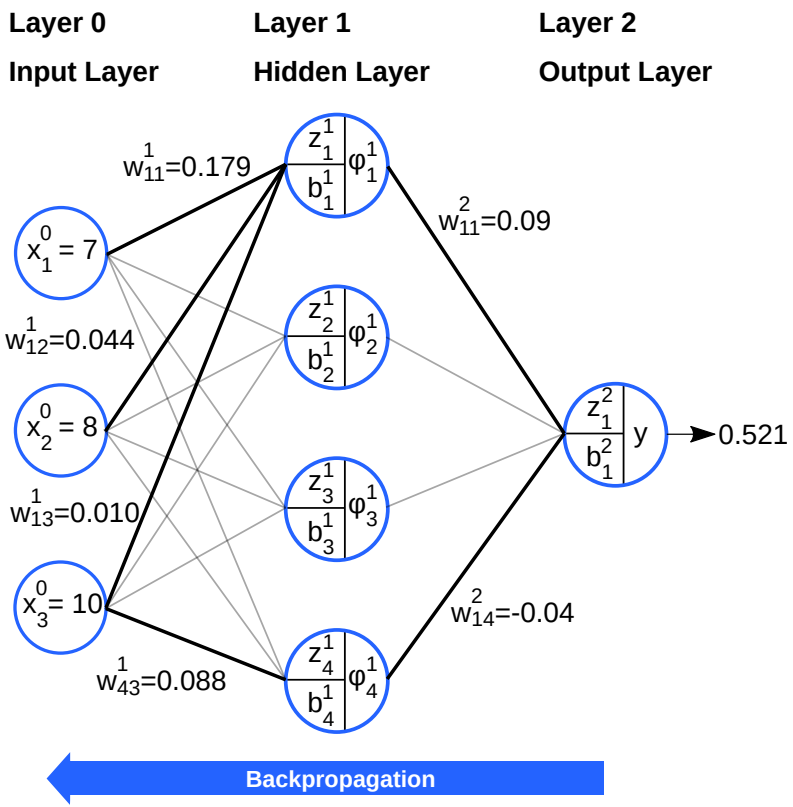
We get:

$$\frac{\partial y}{\partial z_1^2} = \frac{\partial}{\partial z_1^2}\varphi(z_1^2) = (1-y)y$$

Backpropagation in a Multilayer Neural Network

We can now start **Updating Parameters on the Output-Hidden Layer**. Backpropagation works backwards from the output layer to layer 1.

As an example, we update w_{11}^2 and b_1^2.



We first calculate the derivatives of the **weights**.

By chain rule of differentiation, we have

$$\frac{\partial E}{\partial w_{11}^2} = \frac{\partial E}{\partial y}\frac{\partial y}{z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2}, \quad \frac{\partial z_1^2}{\partial w_{11}^2} = \varphi_1^1,$$

$$\frac{\partial y}{\partial z_1^2} = (1-y)y$$

Next, the derivative of cross-entropy loss function:

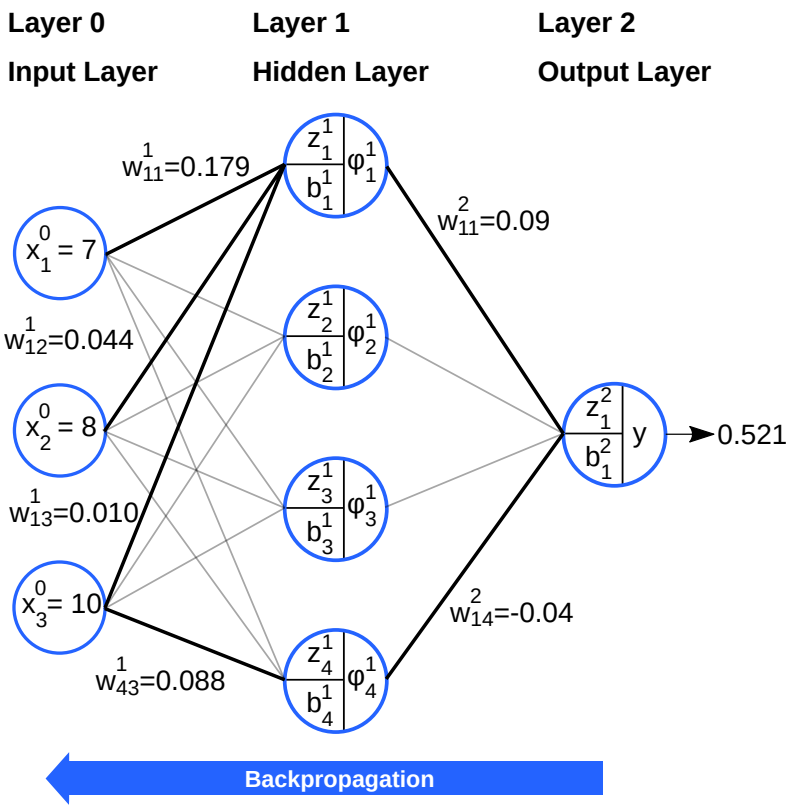
$$\frac{\partial E}{\partial y} = \frac{\partial}{\partial y}(-[\hat{y}\ln y + (1-\hat{y})\ln(1-y)])$$

$$= -\frac{\hat{y}}{y} + \frac{1-\hat{y}}{1-y}$$

Backpropagation in a Multilayer Neural Network

We can now start **Updating Parameters on the Output-Hidden Layer**. Backpropagation works backwards from the output layer to layer 1.

As an example, we update w_{11}^2 and b_1^2.



We first calculate the derivatives of the **weights**.

By chain rule of differentiation, we have

$$\frac{\partial E}{\partial w_{11}^2} = \frac{\partial E}{\partial y}\frac{\partial y}{z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2}, \quad \frac{\partial z_1^2}{\partial w_{11}^2} = \varphi_1^1,$$

$$\frac{\partial y}{\partial z_1^2} = (1-y)y, \quad \frac{\partial E}{\partial y} = -\frac{\hat{y}}{y} + \frac{1-\hat{y}}{1-y}$$

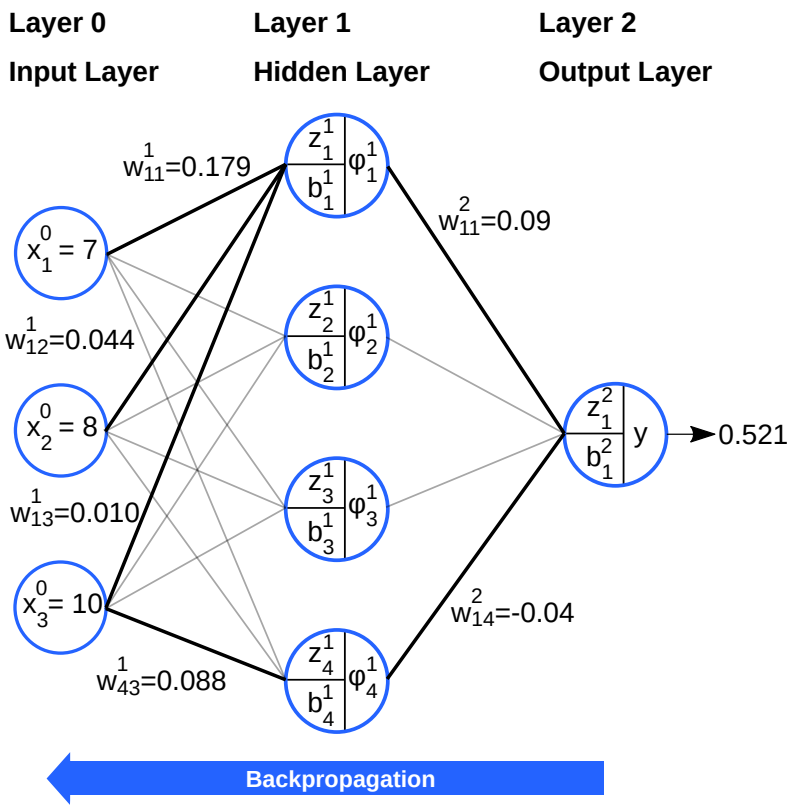
Therefore,

$$\frac{\partial E}{\partial w_{11}^2} = (y-\hat{y})\varphi_1^1 = -0.405$$

Backpropagation in a Multilayer Neural Network

We can now start **Updating Parameters on the Output-Hidden Layer**. Backpropagation works backwards from the output layer to layer 1.

As an example, we update w_{11}^2 and b_1^2.



We now calculate the derivatives of the **bias**.

We used the same arguments as before, that all other variables except b_1^2 are treated as constants therefore on when differentiated they reduce 0.

$$\frac{\partial E}{\partial b_1^2} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial z^2}\frac{\partial z^2}{\partial b_1^2}$$
$$= -\frac{\hat{y}}{y} + \frac{1-\hat{y}}{1-y}(1-y)y$$
$$= (y - \hat{y}) = -0.479$$

Backpropagation in a Multilayer Neural Network

So far, we have computed the gradients with respect to all the parameters at the output-hidden layers.

We can now update all the weights and biases at the output-input layers.

Backpropagation in a Multilayer Neural Network

So far, we have computed the gradients with respect to all the parameters at the output-hidden layers.

We can now update all the weights and biases at the output-input layers.

Gradients at the output-input

	i	j=1
$\frac{\partial E}{\partial w_{1i}^1}$	1	-0.405
	2	-0.063
	3	-0.169
	4	-0.181
$\frac{\partial E}{\partial b_1^2}$		-0.479

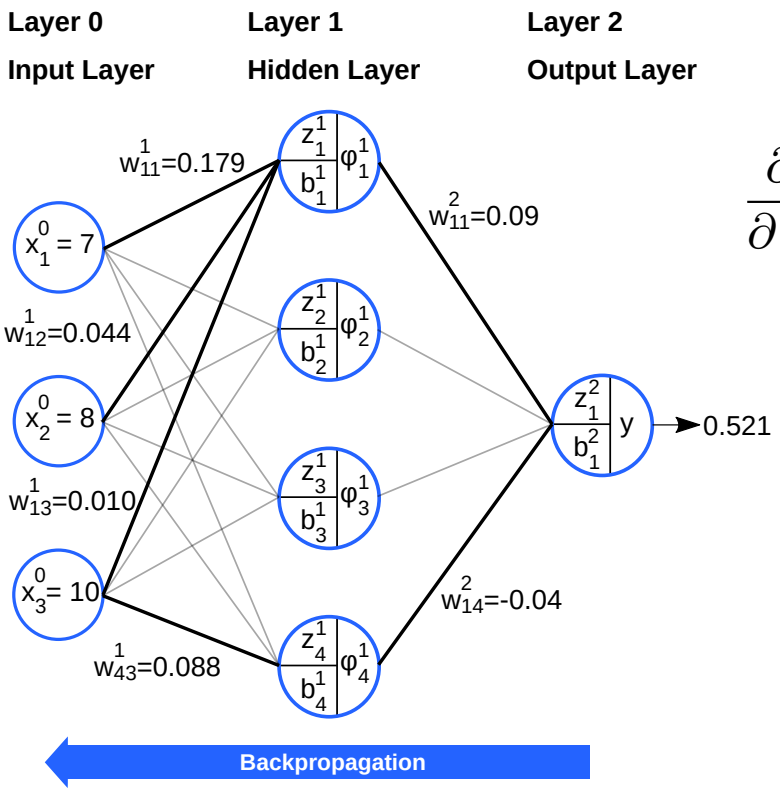
As example, update w_{13}^1 and b_3^1:

$$w_{13}^1 = w_{13}^1 - \eta \frac{\partial E}{\partial w_{13}^1}$$
$$= 0.01 - 0.05(-0.055)$$
$$= 0.0375$$

$$b_3^1 = b_1^1 3 - \eta \frac{\partial E}{\partial b_3^1}$$
$$= 0 - 0.05(-0.001)$$
$$= 0.00005$$

Backpropagation in a Multilayer Neural Network

We can now start **Updating Parameters on the Hidden-Input Layer**.
Backpropagation works backwards from the output layer to layer 1.
As before, we need derivatives of E with respect to all the weights and biases at these layers. We have $4 \times 1 = 4$ weights and 1 bias at the output-hidden layers.
As example, we update w_{43}^1 and b_2^1.



$$\frac{\partial E}{\partial w_{43}^1} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial z_1^2}\frac{\partial z_1^2}{\partial \varphi_4^1}\frac{\partial \varphi_4^1}{\partial z_4^1}\frac{\partial z_4^1}{\partial w_{43}^1}$$
$$= (-\frac{\hat{y}}{y} + \frac{1-\hat{y}}{1-y})(1-y)yw_{14}^2(1-f_4^1)x_3^0$$
$$= (y-\hat{y})w_{14}^2(1-f_4^1)f_4^1x_3^0$$
$$= 0.045$$

Backpropagation in a Multilayer Neural Network

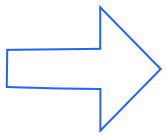
We now have calculated the updated parameters for all the layers using back-propagation algorithm.
When we now run the next forward pass with these updated parameters, we get a new model prediction of $y = 0.648$, up from the previous value of 0.521.

$$y_0 = 0.521$$
$$y_1 = 0.648$$
$$y_2 = 0.758$$
$$y_3 = 0.836$$
$$y_4 = 0.881$$
$$y_5 = 0.908$$
$$y_6 = 0.925$$
$$....$$

Backpropagation in a Multilayer Neural Network

We now have calculated the updated parameters for all the layers using back-propagation algorithm.
When we now run the next forward pass with these updated parameters, we get a new model prediction of $y = 0.648$, up from the previous value of 0.521.

$$y_0 = 0.521$$
$$y_1 = 0.648$$
$$y_2 = 0.758$$
$$y_3 = 0.836$$
$$y_4 = 0.881$$
$$y_5 = 0.908$$
$$y_6 = 0.925$$
$$....$$



The model is **learning** as it's moving closer to $\hat{y} = 1$.

Backpropagation in a Multilayer Neural Network

Some important **Terminology** for training Neural Networks:

One **epoch** is when the entire dataset is passed through the network once. One epoch comprises of one instance of a forward pass and backpropagation.

The **batch size** is the number of training examples passed through the network simultaneously. In the example, we have one training example. In cases where we have a large dataset, the data can be passed through the network in batches.

One **iteration** equals one pass using training examples set as batch size. One pass is a forward pass and a back-propagation.

Backpropagation in a Multilayer Neural Network

Some important **Terminology** for training Neural Networks:

One **epoch** is when the entire dataset is passed through the network once. One epoch comprises of one instance of a forward pass and backpropagation.

The **batch size** is the number of training examples passed through the network simultaneously. In the example, we have one training example. In cases where we have a large dataset, the data can be passed through the network in batches.

One **iteration** equals one pass using training examples set as batch size. One pass is a forward pass and a back-propagation.

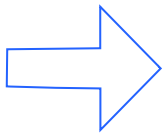
example:
If we have 2000 training examples and set batch size of 20, then it takes 100 iterations to complete 1 epoch.

Backpropagation in a Multilayer Neural Network

When creating Neural Networks, one has to ask: how many hidden layers, and how many neurons in each layer are necessary?

Backpropagation in a Multilayer Neural Network

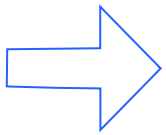
When creating Neural Networks, one has to ask: how many hidden layers, and how many neurons in each layer are necessary?



Neural Network Architecture

Backpropagation in a Multilayer Neural Network

When creating Neural Networks, one has to ask: how many hidden layers, and how many neurons in each layer are necessary?



Neural Network Architecture

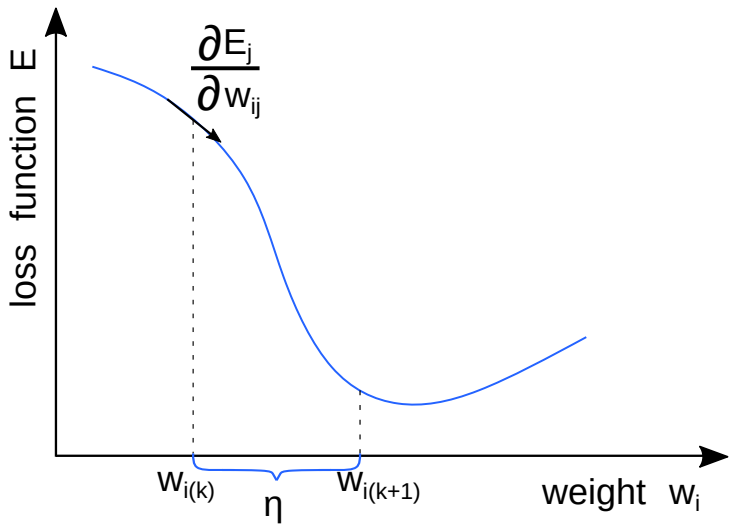
Some thoughts:

- The number of neurons in the first hidden layer creates as many linear decision boundaries to classify the original data. It is not helpful (in theory) to create a deeper neural network if the first layer doesn't contain the necessary number of neurons.
- Add flexibility if it improves the performance, up until overfitting occurs.
- Suggestions on the number of neurons per layer (from astroML): number should be less than twice the size of input nodes; number should be between the number of input and output nodes

The Role of the Loss Function

Loss function: A function of the difference between expected outputs $\hat{y}$ vs. training outputs y, generally written as $E_j = \frac{1}{2} e_j^2 = \frac{1}{2}(\hat{y}_j - y_j)^2$. We can use it to determine how to update weights and biases to reduce the error.

Cost function: Sometimes used to make clear whether we are talking about loss regarding a single training example or the entire training data set. When using this terminology: The loss function computes the error for a single training example, while the cost function (loss over the data set) is the average of the loss functions of the entire training set. The optimization strategies aim at *minimizing the cost function*.



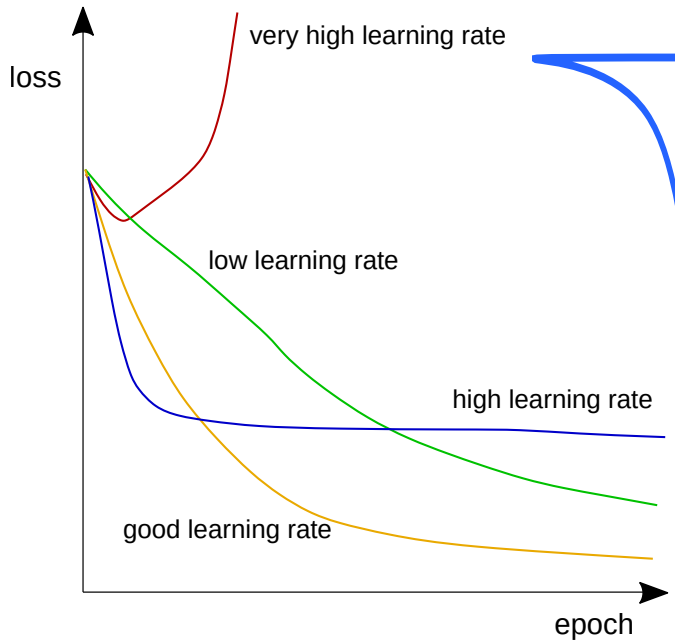
The Role of the Loss Function

The equation

$$w_{ij(k+1)} = w_{ij(k)} - \eta \frac{\partial E_j}{\partial w_{ij}}$$

is an expression for finding an updated weight $w_{ij(k+1)}$ of perceptron j using the loss function E and a learning rate η.

Some intuition on the loss and learning rate:



One **epoch** is when the entire dataset is passed through the network once. One epoch comprises of one instance of a forward pass and backpropagation.

The Loss Function

Given a dataset of examples $\{(y_i, \hat{y}_i)\}_{i=1}^{N}$ where y_i is the classification output and $\hat{y}_i$ is the (integer) label, the loss over the data set is a sum of losses over the examples:

$$E = \frac{1}{N} \sum_{i=1}^{n} E_i(y_i, \hat{y}_i)$$

The Loss Function

Given a dataset of examples $\{(y_i, \hat{y}_i)\}_{i=1}^{N}$ where y_i is the classification output and $\hat{y}_i$ is the (integer) label, the loss over the data set is a sum of losses over the examples:

$$E = \frac{1}{N} \sum_{i=1}^{n} E_i(y_i, \hat{y}_i)$$

In the example in the beginning we used a binary cross-entropy loss function.

The Loss Function

Given a dataset of examples $\{(y_i, \hat{y}_i)\}_{i=1}^{N}$ where y_i is the classification output and $\hat{y}_i$ is the (integer) label, the loss over the data set is a sum of losses over the examples:

$$E = \frac{1}{N} \sum_{i=1}^{n} E_i(y_i, \hat{y}_i)$$

In the example in the beginning we used a binary cross-entropy loss function.

How to chose a loss function?

The Loss Function

loss functions in classification processes

classification tasks: predict categorial variables (the labels) instead of continues ones (as in regression).

The Loss Function

loss functions in classification processes

classification tasks: predict categorial variables (the labels) instead of continues ones (as in regression).

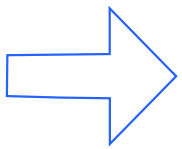
To predict the categorial variables we make use of **probability theory**. Suppose our machine-learning classification model says that an object x belongs to class A with a probability of 0.9. Thus x will be classified as belonging to A. If the true class label is also A, the error will be 0. If the model misclassfied it and the true class is $\neg A$, the error will be 1.

The Loss Function

loss functions in classification processes

classification tasks: predict categorial variables (the labels) instead of continues ones (as in regression).

To predict the categorial variables we make use of **probability theory**. Suppose our machine-learning classification model says that an object x belongs to class A with a probability of 0.9. Thus x will be classified as belonging to A. If the true class label is also A, the error will be 0. If the model misclassfied it and the true class is $\neg A$, the error will be 1.



this error calculation always gives us discrete values, making the loss function undifferentiable

The Loss Function

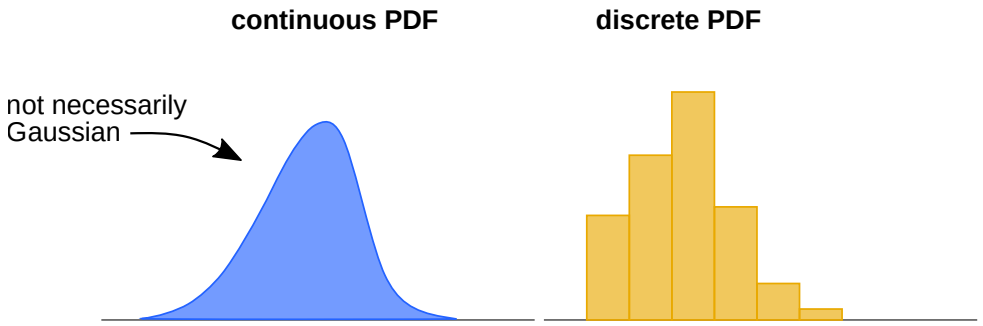
loss functions in classification processes

We can **solve** this by treating the predicted probabilities as coming from one probability densitiy function (PDF) and the actual probabilities from another. The objective is making these two PDFs match.

Entropy signifies uncertainty. For a random variable x coming from a distribution $p(x)$, **entropy** S is defined as:

$$S = \begin{cases} -\int p(x)\ln(p(x))dx, & \text{if } x \text{ is continuous} \\ -\sum_x p(x)\ln(p(x)), & \text{if } x \text{ is discrete.} \end{cases}$$

If the entropy is larger, we are less sure about x being from $p(x)$. For a lower entropy, the confidence will be higher.



The Loss Function

loss functions in classification processes

Binary Cross-Entropy Loss:

Only two classes are present. If the probability of x being in class A is p, then the probability of x being in class $B = \neg A$ will be $(1 - p)$.
The cross-entropy loss, also known as the log loss, with y_i being $p(x = A)$:

$$E_i(\theta) = -[\hat{y}_i \ln(y_i) + (1 - \hat{y}_i) \ln(1 - y_i)] = \begin{cases} -\ln(y_i), & \text{if } \hat{y}_i = 1 \\ -\ln(1 - y_i), & \text{if } \hat{y}_i = 0. \end{cases}$$

$$\Rightarrow E = -\frac{1}{n} \sum \hat{y}_i \ln(y_i) + (1 - \hat{y}_i) \ln(1 - y_i)$$

The Loss Function

loss functions in classification processes

Binary Cross-Entropy Loss:

Only two classes are present. If the probability of x being in class A is p, then the probability of x being in class $B = \neg A$ will be $(1 - p)$.
The cross-entropy loss, also known as the log loss, with y_i being $p(x = A)$:

$$E_i(\theta) = -[\hat{y}_i \ln(y_i) + (1 - \hat{y}_i) \ln(1 - y_i)] = \begin{cases} -\ln(y_i), & \text{if } \hat{y}_i = 1 \\ -\ln(1 - y_i), & \text{if } \hat{y}_i = 0. \end{cases}$$

$$\Rightarrow E = -\frac{1}{n} \sum \hat{y}_i \ln(y_i) + (1 - \hat{y}_i) \ln(1 - y_i)$$

To make sure the probabilities given by the y_i sum up to 1, a **sigmoid activation function** was used in the last layer.

The Loss Function

loss functions in classification processes

Multi-Class Cross-Entropy Loss:

For multi-class classification, the truth is a vector with 1 for a correct answer and 0 for others. The entropy calculation will be the same, but the vector $\hat{y}$ will be represented as a **vector one-hot encoded**.

The Loss Function

loss functions in classification processes

Multi-Class Cross-Entropy Loss:

For multi-class classification, the truth is a vector with 1 for a correct answer and 0 for others. The entropy calculation will be the same, but the vector $\hat{y}$ will be represented as a **vector one-hot encoded**.

example: suppose there are three classes A, B, C, then $\hat{y}_A = [1, 0, 0]$, $\hat{y}_B = [0, 1, 0]$, $\hat{y}_C = [0, 0, 1]$.

The Loss Function

loss functions in classification processes

Multi-Class Cross-Entropy Loss:

For multi-class classification, the truth is a vector with 1 for a correct answer and 0 for others. The entropy calculation will be the same, but the vector $\hat{y}$ will be represented as a **vector one-hot encoded**.

example: suppose there are three classes A, B, C, then $\hat{y}_A = [1, 0, 0]$, $\hat{y}_B = [0, 1, 0]$, $\hat{y}_C = [0, 0, 1]$.

we then get:

$$E_i = -\sum_{j=1}^{c} (\hat{y}_{ij} \ln(y_{ij}))$$

where c are the classes, $\hat{y}_{ij}$ is in **one-hot vector representation**, y_{ij} is the predicted probability for being in class j when the i-th input x_i is provided.

To make sure the probabilities given by the y sum up to 1, a **softmax activation function** was used in the last layer.

The Loss Function

loss functions in regression processes

regression tasks: predict continuous variables instead of categorial ones (as in classification)

Suppose we are trying to fit the function f using machine learning on the training data $\mathbf{x} = \{x_1, ..., x_n\}$ so that $f(\mathbf{x})$ fits $\hat{\mathbf{y}} = \{y_1, ..., y_n\}$. But this function f cannot be perfect, and there will be errors. To quantify these, we use a loss function.

The Loss Function

loss functions in regression processes

Mean Absolute Error:

The Mean Absolute Error (MAE), also called L-1 loss, is used for regression tasks.

$$E_i = |y_i - \hat{y}_i|$$

$$\Rightarrow E = \frac{1}{n} \sum_{i}^{n} |y_i - \hat{y}_i|$$

The Loss Function

loss functions in regression processes

Mean Absolute Error:

The Mean Absolute Error (MAE), also called L-1 loss, is used for regression tasks.

$$E_i = |y_i - \hat{y}_i|$$

$$\Rightarrow E = \frac{1}{n} \sum_{i}^{n} |y_i - \hat{y}_i|$$

limitations of MAE:
Although frequently used, it is a non-differentiable function, so finding gradients to update the parameters is more difficult. Near the minimum, the gradients become undefined, making MAE unstable.

The Loss Function

loss functions in regression processes

Mean Squared Error:

Mean Squared Error (MSE), also called L-2 loss, is calculated by taking the mean of squared differences between actual and predicted values.

$$E_i = (y_i - \hat{y}_i)^2$$

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

As a quadratic equation with a global minimum and no local minima, gradient descent does not fall into the *local minimum trap* and parameter optimization will work for guaranteed. Thus, MSE is one of the most favorable loss functions for regression tasks.

The Loss Function

loss functions in regression processes

Mean Squared Error:

Mean Squared Error (MSE), also called L-2 loss, is calculated by taking the mean of squared differences between actual and predicted values.

$$E_i = (y_i - \hat{y}_i)^2$$

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

As a quadratic equation with a global minimum and no local minima, gradient descent does not fall into the *local minimum trap* and parameter optimization will work for guaranteed. Thus, MSE is one of the most favorable loss functions for regression tasks.

limitations of MSE:
The difference between actual value $\hat{y}_i$ and predicted value y_i will be higher. MSE is less robust to the presence of outliers, and it is advised to not use MSE when there are too many outliers in the dataset.

The Loss Function

loss functions in regression processes

Huber Loss:

MAE and MSE are prevalent, but come with limitations:

- MAE is more robust to outliers than MSE - when the difference is higher, MAE is more stable than MSE
- MSE is in general more stable than MAE, especially when the difference between prediction and actual value is small.

The Huber loss takes the best from both MAE and MSE, avoiding their shortcomings.

The Loss Function

loss functions in regression processes

Huber Loss:

quadratic

$$E_i = \begin{cases} \frac{1}{2}(\hat{y}_i - y_i)^2, & \text{if } |\hat{y}_i - y_i| < \delta \\ \delta|\hat{y}_i - y_i| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

linear

The Loss Function

loss functions in regression processes

Huber Loss:

quadratic

$$E_i = \begin{cases} \frac{1}{2}(\hat{y}_i - y_i)^2, & \text{if } |\hat{y}_i - y_i| < \delta \\ \delta|\hat{y}_i - y_i| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

linear

As defined above, the Huber loss function is strongly convex in a uniform neighborhood of its minimum $\hat{y}_i - y_i = 0$; at the boundary of this uniform neighborhood, the Huber loss function has a differentiable extension to an affine function at points $\hat{y}_i - y_i = \pm\delta$.

These properties allow it to combine much of the sensitivity of the mean-unbiased, minimum-variance estimator of the mean (using the quadratic loss function) and the robustness of the median-unbiased estimator (using the absolute value function).

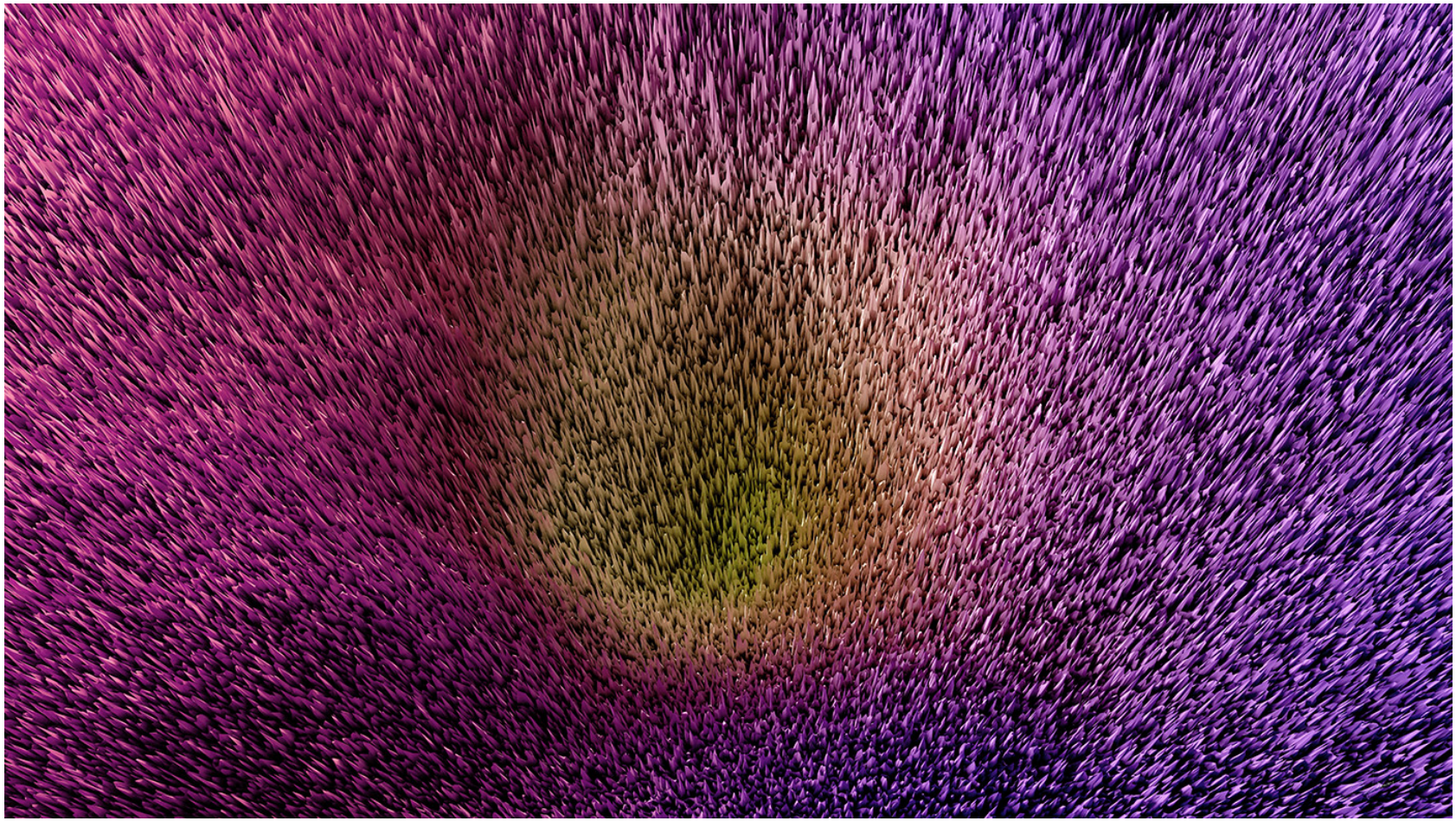
Loss Functions and Activations

some common choices:

problem type	last-layer activation	loss function
binary classification	sigmoid	binary cross-entropy
multiclass, single-label classification	softmax	categorical cross-entropy
multiclass, multi-label classification	sigmoid	binary cross-entropy
regression of arbitrary values	None	mean-square error (MSE)
regression of values between 0 and 1	sigmoid	MSE or binary cross-entropy

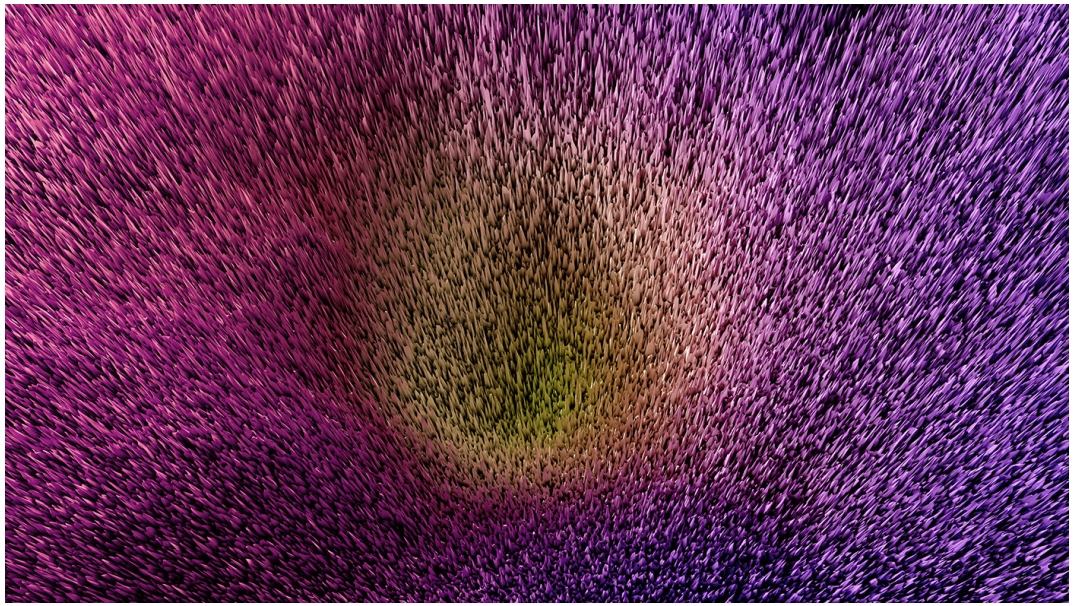
Gradient Descent

cost function (loss over the data set) can become very complex



Gradient Descent

cost function (loss over the data set) can become very complex



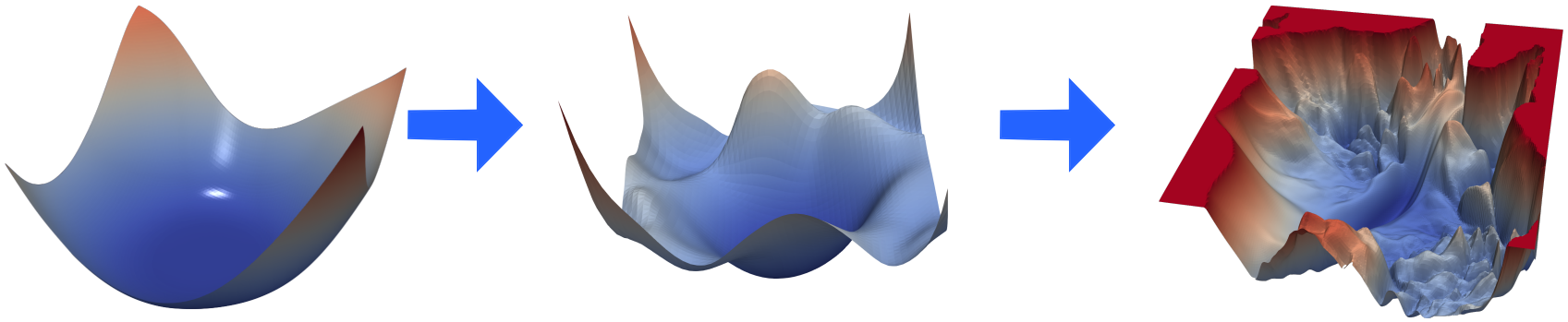
Loss Landscape created from the training process of a convolutional neural network.

credit:

`https://www.cs.umd.edu/~tomg/projects/landscapes/`

`https://losslandscape.com/gallery/`

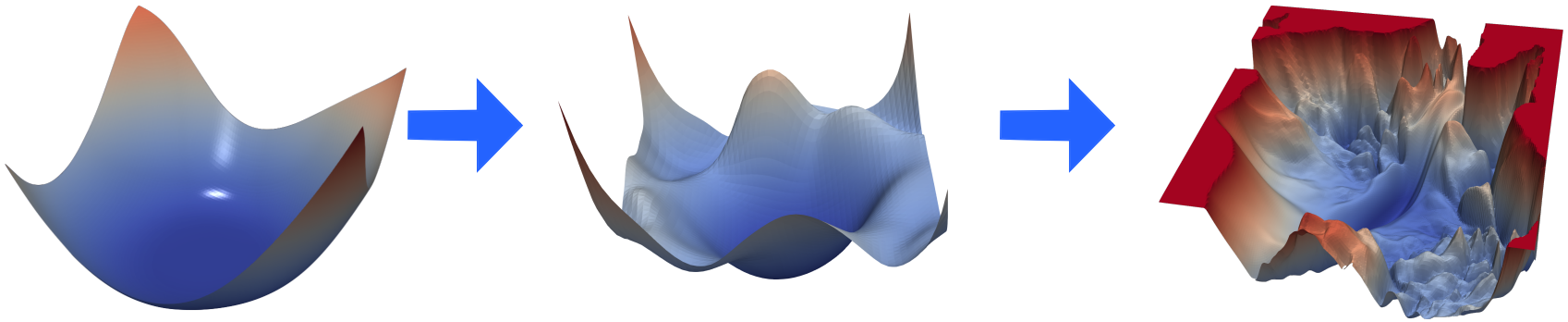
Gradient Descent



network architecture has a dramatic effect on the loss landscape:

Shallow networks have smooth landscapes populated by wide, convex regions. However, as networks become deeper, landscapes spontaneously become chaotic and highly non-convex, leading to poor training behavior.

Gradient Descent



network architecture has a dramatic effect on the loss landscape:

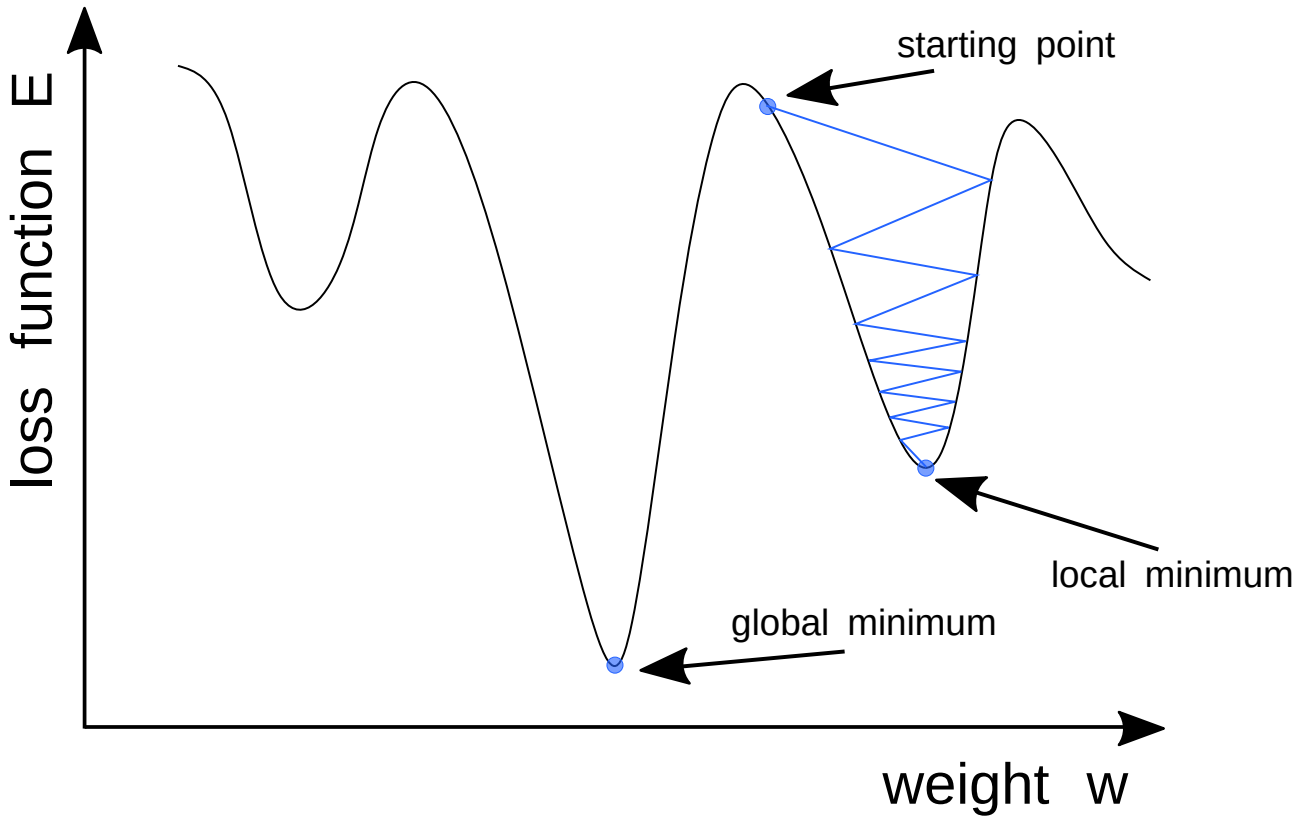
Shallow networks have smooth landscapes populated by wide, convex regions. However, as networks become deeper, landscapes spontaneously become chaotic and highly non-convex, leading to poor training behavior.

⇨ Gradient Descent needs to explore these landscapes

Recap: Gradient Descent

training neural networks is a **non-convex optimization problem**

this means we can run into many local minima during training



Recap: Gradient Descent

the simplest algorithm that attempts to minimize the loss $E(\theta)$ is Gradient Descent:

Recap: Gradient Descent

the simplest algorithm that attempts to minimize the loss $E(\boldsymbol{\theta})$ is Gradient Descent:

1. initialize the weights and biases $\boldsymbol{\theta}$ randomly.
2. update $\boldsymbol{\theta}_k \rightarrow \boldsymbol{\theta}_{k+1}$:

$$\mathbf{v}_k = \eta \nabla_\theta E(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

with the learning rate η and step k

repeat until the difference in values of E between two consecutive iterations goes below a pre-defined threshold

Recap: Gradient Descent

the simplest algorithm that attempts to minimize the loss $E(\boldsymbol{\theta})$ is Gradient Descent:

1. initialize the weights and biases $\boldsymbol{\theta}$ randomly.
2. update $\boldsymbol{\theta}_k \rightarrow \boldsymbol{\theta}_{k+1}$:

$$\mathbf{v}_k = \eta \nabla_\theta E(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

with the learning rate η and step k

repeat until the difference in values of E between two consecutive iterations goes below a pre-defined threshold

- requires a careful choice of learning rate
- sensitive to initial conditions
- can become stuck in local minima
- long time to escape saddle points
- learning rate is constant

Stochastic Gradient Descent

Problem with Gradient Descent: For a single update of each parameter, we go over N data points (summation $i = 0$ to N). Datasets involving thousands of features and millions of data points are not uncommon.

Stochastic Gradient Descent

Problem with Gradient Descent: For a single update of each parameter, we go over N data points (summation $i = 0$ to N). Datasets involving thousands of features and millions of data points are not uncommon.

Each update would require thousands of derivatives $\times$ millions of terms $\rightarrow \sim$ billion calculations per step

Stochastic Gradient Descent

Problem with Gradient Descent: For a single update of each parameter, we go over N data points (summation $i = 0$ to N). Datasets involving thousands of features and millions of data points are not uncommon.

Each update would require thousands of derivatives $\times$ millions of terms $\rightarrow \sim$ billion calculations per step

Solution:

In Stochastic Gradient Descent (SGD), stochasticity is added by approximating the gradient on a data subset called a **mini batch** (typically in the order of 10 - 100 data points)

full iteration using all mini batches of a dataset is called an epoch

update rule is now just

$$\mathbf{v}_k = \eta_k \nabla_\theta E_{mb}(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

Stochastic Gradient Descent

comparing SGD vs. GD

The main difference is the number of data points to go through before each update: which is 1 in case of SGD vs. all in case of GD.

SGD:

decreases the chance of becoming stuck in local minima

computation is fast as we take one data point at a time

⊕ as considering only one data point for updating parameters, SGD is heavily affected by outliers - encountering an outlier requires extra updates to get back on track of convergence

less overfitting as it is more generalized and randomized

⊖ it takes much more time to converge as the gradient we get in SGD is not the true Gradient but just an approximation

Mini-Batch Gradient Descent

This variation is a **compromise** between both GD and SGD:

In mini-batch GD parameters are updated based on small samples of size $1 < b < N$. So instead of taking a single data point (case of SGD) or the complete dataset (case of GD), randomly b data points were taken from the data set for calculating the gradient and updating parameters.

Mini-Batch Gradient Descent

This variation is a **compromise** between both GD and SGD:

In mini-batch GD parameters are updated based on small samples of size $1 < b < N$. So instead of taking a single data point (case of SGD) or the complete dataset (case of GD), randomly b data points were taken from the data set for calculating the gradient and updating parameters.

Mini-Batch GD usually converges a little faster (in terms of iterations) than SGD as the gradient approximation in mini-batch GD is closer to the true gradient as compared to that in SGD.

As there are chances that data points can be repeated in different batches, mini-batch GD is more prone to overfitting as compared to SGD.

Mini-Batch Gradient Descent

This variation is a **compromise** between both GD and SGD:

In mini-batch GD parameters are updated based on small samples of size $1 < b < N$. So instead of taking a single data point (case of SGD) or the complete dataset (case of GD), randomly b data points were taken from the data set for calculating the gradient and updating parameters.

⊕ Mini-Batch GD usually converges a little faster (in terms of iterations) than SGD as the gradient approximation in mini-batch GD is closer to the true gradient as compared to that in SGD.

⊖ As there are chances that data points can be repeated in different batches, mini-batch GD is more prone to overfitting as compared to SGD.

The size of the mini-batch b depends on the size of the dataset. But generally, it should be around 1.5 times the number of outliers so that even in the worst case, it won't be affected much by outliers.

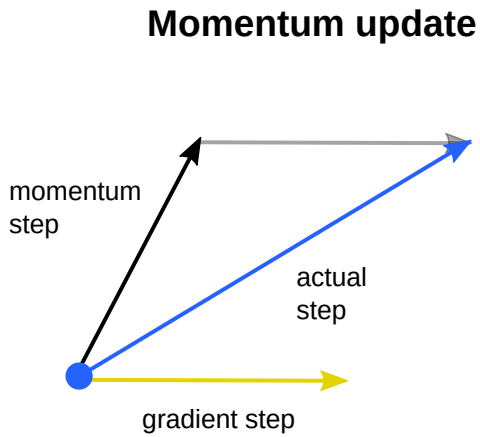
Stochastic Gradient Descent with Momentum

SGD with momentum adds an **inertia term** that retains some memory of the direction:

$$\mathbf{v}_k = \gamma \mathbf{v}_{k-1} + \eta_k \nabla_\theta E_m(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

with the momentum parameter $\gamma \sim 0.9$

SGD with momentum helps parameter updates gain speed in persistent smaller gradients while suppressing oscillatory high gradients



Stochastic Gradient Descent with Momentum

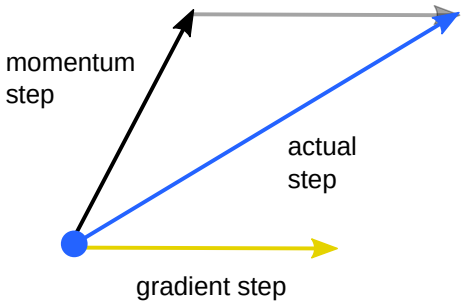
The **Nesterov accelerated momentum** (NAG) modifies this concept: First jump in the direction of the previous accumulated gradient, then measure the gradient where you end up and make a correction. This is done by performing the update using the partial derivative of the projected update rather than the derivative current variable value.

jump correction

$$\mathbf{v}_k = \gamma \mathbf{v}_{k-1} + \eta_k \nabla_\theta E_m(\boldsymbol{\theta}_k - \gamma \mathbf{v}_{k-1})$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

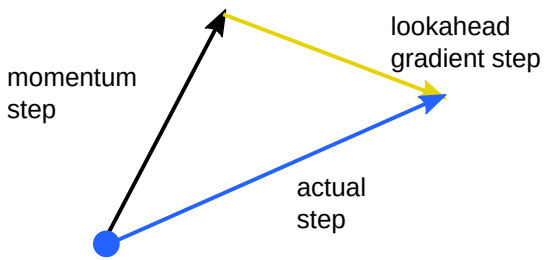
Momentum update

$\mathbf{v}_k = \gamma \mathbf{v}_{k-1} + \eta_k \nabla_\theta E_m(\boldsymbol{\theta}_k)$



Nesterov momentum update

$\mathbf{v}_k = \gamma \mathbf{v}_{k-1} + \eta_k \nabla_\theta E_m(\boldsymbol{\theta}_k - \gamma \mathbf{v}_{k-1})$



This anticipatory update results in increased responsiveness, which significantly increases the performance.

Gradient Descent - Summary

Batch gradient descent: All available data is injected at once. This version implies a high risk of getting stuck, since the gradient will be calculated using all the samples, and the variations will become minimal.

Gradient Descent - Summary

Batch gradient descent: All available data is injected at once. This version implies a high risk of getting stuck, since the gradient will be calculated using all the samples, and the variations will become minimal.

Stochastic gradient descent (SGD): In each iteration, the gradient is calculated for a single random sample only. While introducting the desired randomness, the disadvantage is its slowness, as much more iterations are necessary which can't be parallelized.

Gradient Descent - Summary

Batch gradient descent: All available data is injected at once. This version implies a high risk of getting stuck, since the gradient will be calculated using all the samples, and the variations will become minimal.

Stochastic gradient descent (SGD): In each iteration, the gradient is calculated for a single random sample only. While introducting the desired randomness, the disadvantage is its slowness, as much more iterations are necessary which can't be parallelized.

(Stochastic) Mini-batch gradient descent: Instead of single samples, N random items are introduced on each iteration. This preserves the advantages of SGD but enables faster training due to parallelization.

Gradient Descent - Summary

Batch gradient descent: All available data is injected at once. This version implies a high risk of getting stuck, since the gradient will be calculated using all the samples, and the variations will become minimal.

Stochastic gradient descent (SGD): In each iteration, the gradient is calculated for a single random sample only. While introducting the desired randomness, the disadvantage is its slowness, as much more iterations are necessary which can't be parallelized.

(Stochastic) Mini-batch gradient descent: Instead of single samples, N random items are introduced on each iteration. This preserves the advantages of SGD but enables faster training due to parallelization.

SGD with momentum: An inertia term that retains some memory of the direction helps parameter updates gain speed in persistent smaller gradients while suppressing oscillatory high gradients.

Gradient Descent - Summary

Batch gradient descent: All available data is injected at once. This version implies a high risk of getting stuck, since the gradient will be calculated using all the samples, and the variations will become minimal.

Stochastic gradient descent (SGD): In each iteration, the gradient is calculated for a single random sample only. While introducting the desired randomness, the disadvantage is its slowness, as much more iterations are necessary which can't be parallelized.

(Stochastic) Mini-batch gradient descent: Instead of single samples, N random items are introduced on each iteration. This preserves the advantages of SGD but enables faster training due to parallelization.

SGD with momentum: An inertia term that retains some memory of the direction helps parameter updates gain speed in persistent smaller gradients while suppressing oscillatory high gradients.

Nesterov accelerated momentum (NAG): This modifies the concept of SGD with momentum. First a jump in the direction of the previous accumulated gradient is carried out, then the gradient is calculated and a correction carried out. This allows the search to slow down as approaching the optima, reducing the likelihood of missing or overshooting it.

Learning Rate

The learning rate η defines the size of the corrective steps that the model takes to adjust for errors in its output.

$$\mathbf{v}_k = \eta \nabla_\theta E(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

Learning Rate

The learning rate η defines the size of the corrective steps that the model takes to adjust for errors in its output.

$$\mathbf{v}_k = \eta \nabla_\theta E(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

A **high learning rate** shortens the training time, but with lower ultimate accuracy, while a **lower learning rate** takes longer, but with the potential for greater accuracy.

Learning Rate

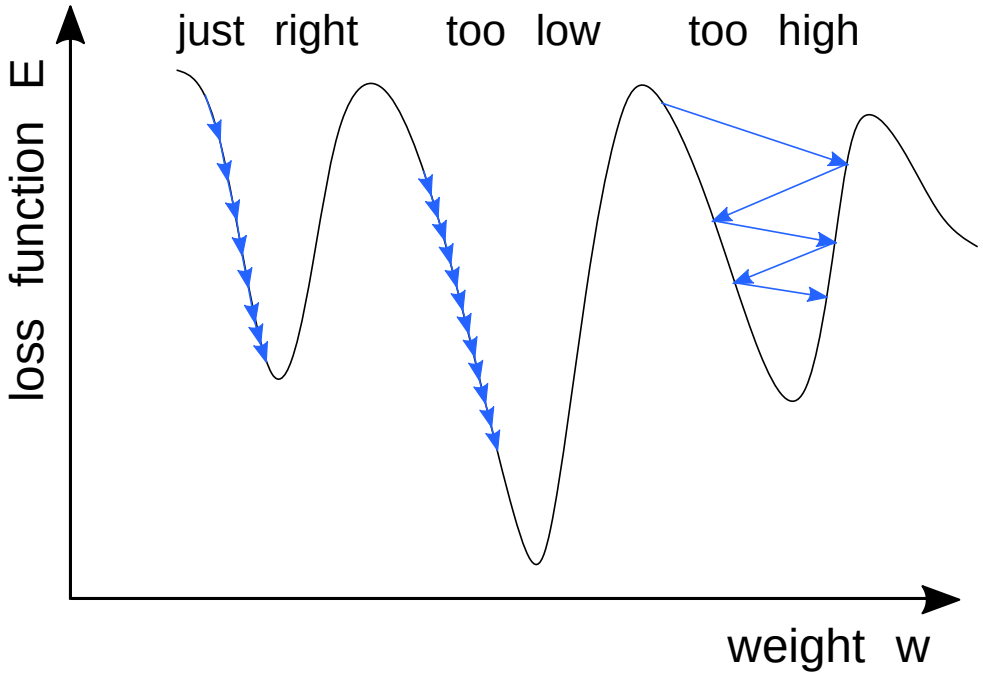
The learning rate η defines the size of the corrective steps that the model takes to adjust for errors in its output.

$$\mathbf{v}_k = \eta \nabla_\theta E(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

A **high learning rate** shortens the training time, but with lower ultimate accuracy, while a **lower learning rate** takes longer, but with the potential for greater accuracy.

- To avoid oscillation inside the network such as alternating weights, and to improve the rate of convergence, an **adaptive learning rate** that increases or decreases as appropriate is used.
- In **reinforcement learning**, the aim is to weight the network to perform actions that minimize long-term (expected cumulative) cost.
- The concept of momentum allows the balance between the gradient and the previous change to be weighted such that the weight adjustment depends to some degree (characterized by the value of the momentum) on the previous change.

Learning Rate



optimal learning rate allows for smoothly reaching the maxima

too small learning rate requires many updates, hence learning is slow

too large learning rate leads to drastic updates with the possibility of overshooting the minima (diverging from it)

Vanishing and Exploding Gradients

As their names suggest, Vanishing and Exploding Gradients refer to the partial derivative of the loss function, trending towards zero and infinity respectively as they are back-propagated (multiplied) to earlier layers.

Vanishing and Exploding Gradients

Vanishing Gradients

During backpropagation, the gradients for the weight update are calculated using the chain rule relying on gradients of later layers:

$$\frac{\partial E}{\partial w_{11}^2} = \frac{\partial E}{\partial y} \frac{\partial y}{z_1^2} \frac{\partial z_1^2}{\partial w_{11}^2} \quad \text{with} \quad \frac{\partial z_1^2}{\partial w_{11}^2} = \frac{\partial}{\partial w_{11}^2}(\varphi_1^1 w_{11}^2 + \varphi_2^1 w_{12}^2 + \varphi_3^1 w_{13}^2 + \varphi_4^1 w_{14}^2 + b_1^2)$$

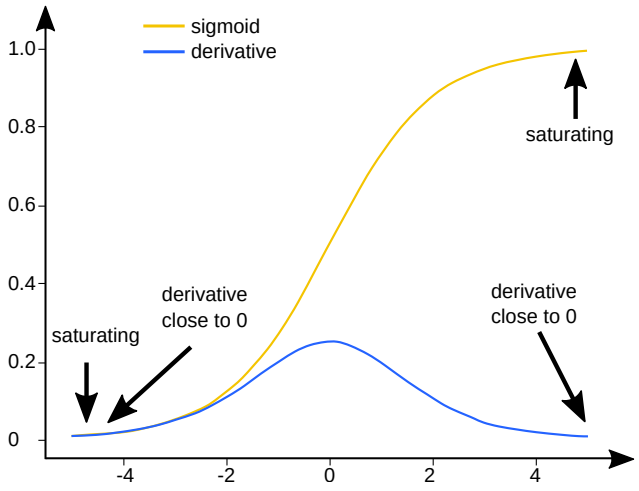
Vanishing and Exploding Gradients

Vanishing Gradients

During backpropagation, the gradients for the weight update are calculated using the chain rule relying on gradients of later layers:

$$\frac{\partial E}{\partial w_{11}^2} = \frac{\partial E}{\partial y}\frac{\partial y}{z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2} \quad \text{with} \quad \frac{\partial z_1^2}{\partial w_{11}^2} = \frac{\partial}{\partial w_{11}^2}(\varphi_1^1 w_{11}^2 + \varphi_2^1 w_{12}^2 + \varphi_3^1 w_{13}^2 + \varphi_4^1 w_{14}^2 + b_1^2)$$

As backpropagation advances from the output layer backwards, the gradients decrease until approaching zero. Updated weights get virtually identical to the old weights, making the model incapable of effective learning especially in the initial layers which learn the data's core features.



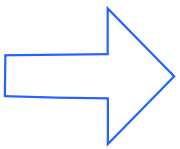
Vanishing and Exploding Gradients

Vanishing Gradients

During backpropagation, the gradients for the weight update are calculated using the chain rule relying on gradients of later layers:

$$\frac{\partial E}{\partial w_{11}^2} = \frac{\partial E}{\partial y}\frac{\partial y}{z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2} \quad \text{with} \quad \frac{\partial z_1^2}{\partial w_{11}^2} = \frac{\partial}{\partial w_{11}^2}(\varphi_1^1 w_{11}^2 + \varphi_2^1 w_{12}^2 + \varphi_3^1 w_{13}^2 + \varphi_4^1 w_{14}^2 + b_1^2)$$

As backpropagation advances from the output layer backwards, the gradients decrease until approaching zero. Updated weights get virtually identical to the old weights, making the model incapable of effective learning especially in the initial layers which learn the data's core features.

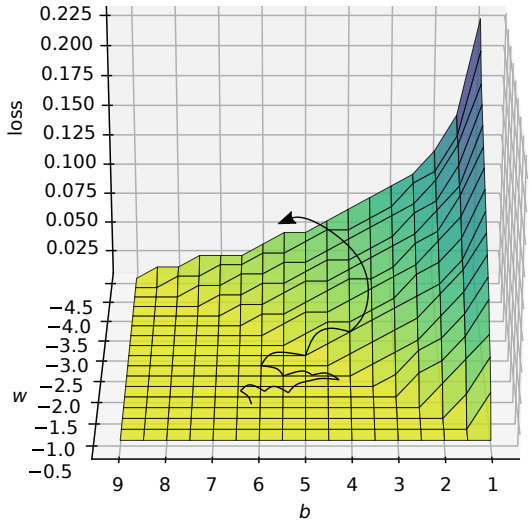


the deeper the neural network (greater number of Hidden Layers), the higher the probability of Vanishing/Exploding Gradients

Vanishing and Exploding Gradients

Exploding Gradients

On the contrary, in some cases the accumulation of large derivatives backwards results in the model being very unstable in training and achieving wildly fluctuating losses, as large changes in the weights creates a very unstable network. This is known as the exploding gradients problem.

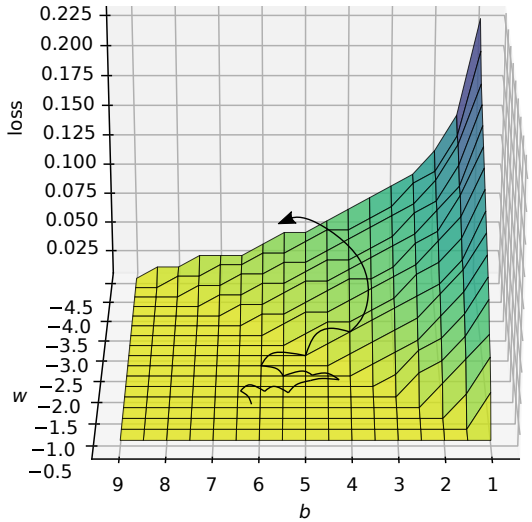


Vanishing and Exploding Gradients

Exploding Gradients

On the contrary, in some cases the accumulation of large derivatives backwards results in the model being very unstable in training and achieving wildly fluctuating losses, as large changes in the weights creates a very unstable network. This is known as the exploding gradients problem.

The problem arises from the initial weight assignment creating large losses. Gradient values can accumulate to the point where large parameter updates happen, causing gradient descent to oscillate without reaching a global minium. Even worse: the weights can get so large that they lead to overflows (NaN values) making further updates impossible.



Vanishing and Exploding Gradients

Following are some signs that can indicate that gradients are vanishing or exploding:

vanishing gradients:

- The parameters of the higher layers change significantly whereas the parameters of lower layers would not change much (or not at all).
- The model weights may become 0 during training.
- The model learns very slowly and perhaps the training stagnates at a very early stage just after a few iterations.

Vanishing and Exploding Gradients

Following are some signs that can indicate that gradients are vanishing or exploding:

vanishing gradients:

- The parameters of the higher layers change significantly whereas the parameters of lower layers would not change much (or not at all).
- The model weights may become 0 during training.
- The model learns very slowly and perhaps the training stagnates at a very early stage just after a few iterations.

exploding gradients:

- There is an exponential growth in the model parameters.
- The model weights may become NaN during training.
- The model experiences avalanche learning.

Vanishing and Exploding Gradients

Fortunately there are some techniques to overcome these issues, apart from using suitable activation functions.

Weight Initialization

Random weight initialization sometimes pose a problem by the very first activation values produced may be slightly too large or too small, planting the seed for vanishing or exploding gradients later on in training.

Weight Initialization

Random weight initialization sometimes pose a problem by the very first activation values produced may be slightly too large or too small, planting the seed for vanishing or exploding gradients later on in training.

Xavier (or Glorot) Initialization

For the proper flow of the signal, X. Glorot & Y. Benigo (2010) argue that:

- The variance of outputs of each layer should be equal to the variance of its inputs.
- The gradients should have equal variance before and after flowing through a layer in the reverse direction.

Weight Initialization

Random weight initialization sometimes pose a problem by the very first activation values produced may be slightly too large or too small, planting the seed for vanishing or exploding gradients later on in training.

Xavier (or Glorot) Initialization

For the proper flow of the signal, X. Glorot & Y. Benigo (2010) argue that:

- The variance of outputs of each layer should be equal to the variance of its inputs.
- The gradients should have equal variance before and after flowing through a layer in the reverse direction.

It is impossible for both conditions to hold for any layer unless the number of inputs to that layer (n_{in}) equals the numbers of neurons in that layer (n_{in}). However, the following compromise works incredibly well in practice: randomly initialize the connection weights for each layer in the network as described in the following.

Weight Initialization

Xavier (or Glorot) Initialization

Suppose we have an input x with n components and a neuron with random weights w and output y.

We know that the variance of $w_i x_i$ is, with expectation value $\mathrm{E}(\cdot)$:

$$\mathrm{Var}(w_i x_i) = \mathrm{E}(x_i)^2 \mathrm{Var}(w_i) + \mathrm{E}(w_i)^2 \mathrm{Var}(x_i) + \mathrm{Var}(w_i)\mathrm{Var}(x_i)$$

We assume that x_i, w_i are identically and independently distributed and get

$$\begin{aligned}\mathrm{Var}(y) &= \mathrm{Var}(w_1 x_1 + ... + w_n x_n) \\ &= \mathrm{Var}(w_1 x_1) + ... + \mathrm{Var}(w_n x_n) \\ &= n\mathrm{Var}(w_i)\mathrm{Var}(x_i)\end{aligned}$$

The variance of the output is the variance of the input scaled by $n\mathrm{Var}(w_i)$. For the variance of y to be equal to the variance of x, $n\mathrm{Var}(w_i)$ should be equal to 1 $\Rightarrow$ the variance of the weight should be $\mathrm{Var}(w_i) = 1/n_{in}$.
with expectation value $\mathrm{E}(\cdot)$.
Similarly, if we go through backpropagation, we apply the same steps and get $\mathrm{Var}(w_i) = 1/n_{out}$.

Weight Initialization

Xavier (or Glorot) Initialization

In order to keep the variance of the input and the output gradient the same, $n_{in} = n_{out}$ must hold. As a compromise, Glorot and Bengio suggest using the average of the n_{in} and n_{out}, proposing that:

$$\mathrm{Var}(w_i) = 1/n_{avg} = \frac{2}{(n_{in} + n_{out})}.$$

This is Xavier Initialization formula to be applied to all weights of a given layer.

Weight Initialization

Xavier (or Glorot) Initialization

In order to keep the variance of the input and the output gradient the same, $n_{in} = n_{out}$ must hold. As a compromise, Glorot and Bengio suggest using the average of the n_{in} and n_{out}, proposing that:

$$\mathrm{Var}(w_i) = 1/n_{avg} = \frac{2}{(n_{in} + n_{out})}.$$

This is Xavier Initialization formula to be applied to all weights of a given layer.

He Initialization

Glorot and Bengio considered a sigmoid activation function, the default choice at that moment. Later on, the sigmoid activation was surpassed by ReLu, because it allowed to solve vanishing / exploding gradients problem. However, it turns out Xavier (Glorot) Initialization is not quite as optimal for ReLU functions. The He initialization solves this with the adjustment of multiplying the variance from the Xavier Initialization by a factor of 2.

Regularization

weight initialization is ony part of the solution:

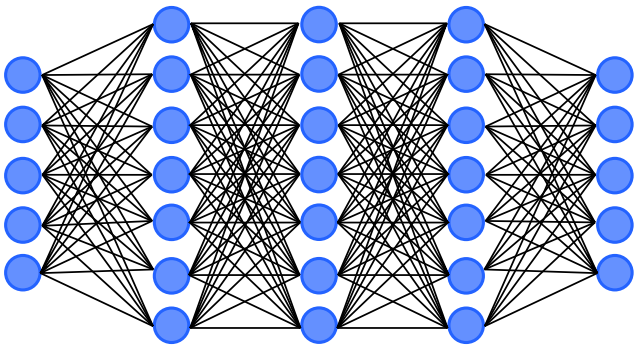
prevent overfitting by limiting the capacity of a neural network with **regularization techniques**

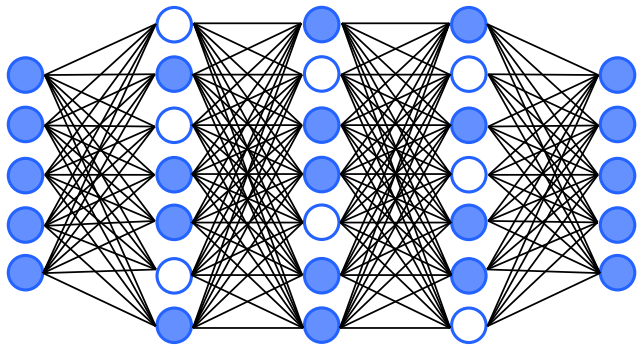
- limit the number of hidden units
- limit the size of weights $\Rightarrow$ **weight decay**
- stop the learning before it has time to overfit $\Rightarrow$ **early stop**

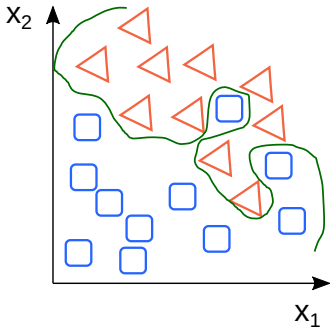
$$E(\theta) = \underbrace{\frac{1}{m}\sum_{i=1}^{m} E_i(y, \hat{y})}_{\text{Data loss: model prediction should match training data}} + \underbrace{\lambda R(\theta)}_{\text{Regularization: model should be simple so it works on test data}}$$

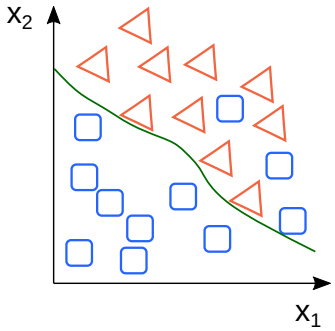
Regularization

Dropout turns off random neurons with probability p for each mini-batch during training (forward pass).
This forces the network to have redundant representation, preventing co-adaptation for features.









model has too many free parameters and over-fits

too many neurons dropped out leading to underfitting

Regularization

Weight decay, or L_2 **regularization**, is a regularization technique applied to the weights of a neural network. We minimize a loss function compromising both the primary loss function and a **penalty term**:

$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$

with the weight decay parameter $\lambda \sim 0.1$.

Regularization

Weight decay, or L_2 **regularization**, is a regularization technique applied to the weights of a neural network. We minimize a loss function compromising both the primary loss function and a **penalty term**:

$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$

with the weight decay parameter $\lambda \sim 0.1$.

This extra term penalizes the squared weights by keeping weights small unless they have big error derivatives.

Regularization

Weight decay, or L_2 **regularization**, is a regularization technique applied to the weights of a neural network. We minimize a loss function compromising both the primary loss function and a **penalty term**:

$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$

with the weight decay parameter $\lambda \sim 0.1$.

This extra term penalizes the squared weights by keeping weights small unless they have big error derivatives.

A modification: L_1 **regularization** using $|w_i|$ instead of w_i^2.

Weight decay can be incorporated directly into the weight update rule, rather than just implicitly by defining it through to error function.

Regularization

Weight decay, or L_2 **regularization**, is a regularization technique applied to the weights of a neural network. We minimize a loss function compromising both the primary loss function and a **penalty term**:

$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$

with the weight decay parameter $\lambda \sim 0.1$.

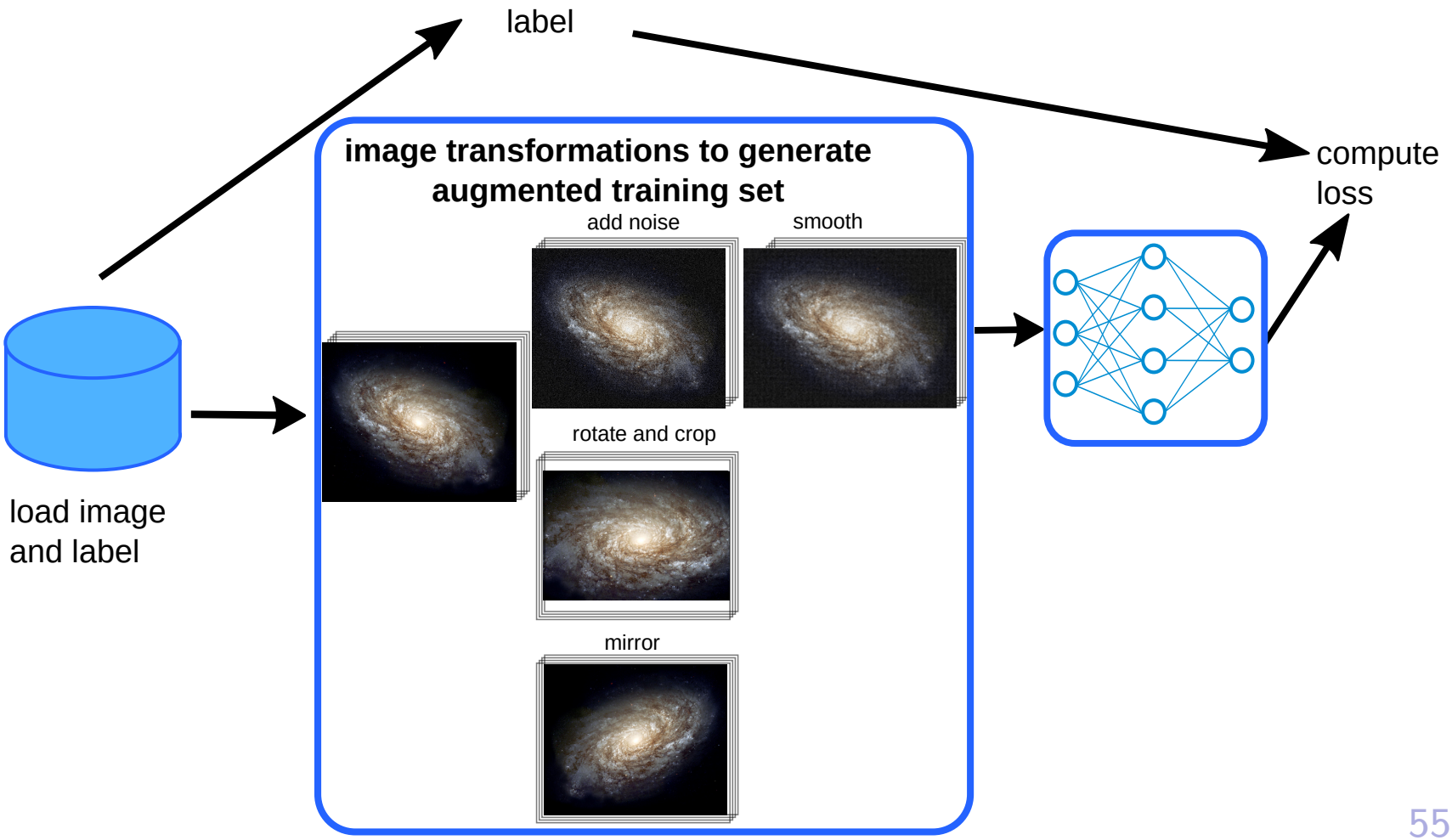
benefits of using weight decay:

- Improved generalization performance of the model by preventing overfitting.
- Reduced complexity of the model by encouraging a sparser learned representation.
- Improved convergence of te optimizer by encouraging smaller weights.

The beauty of weight decay is that the algorithm automatically decides which neurons to knock out in order to lower the complexity.

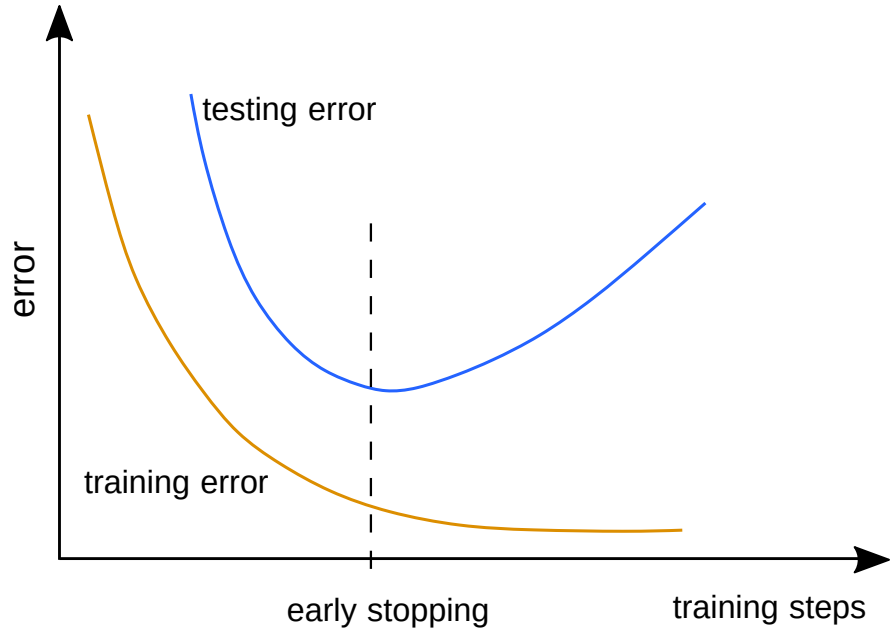
Regularization

the regularization technique of **data augmentation** adds randomness in training



Regularization

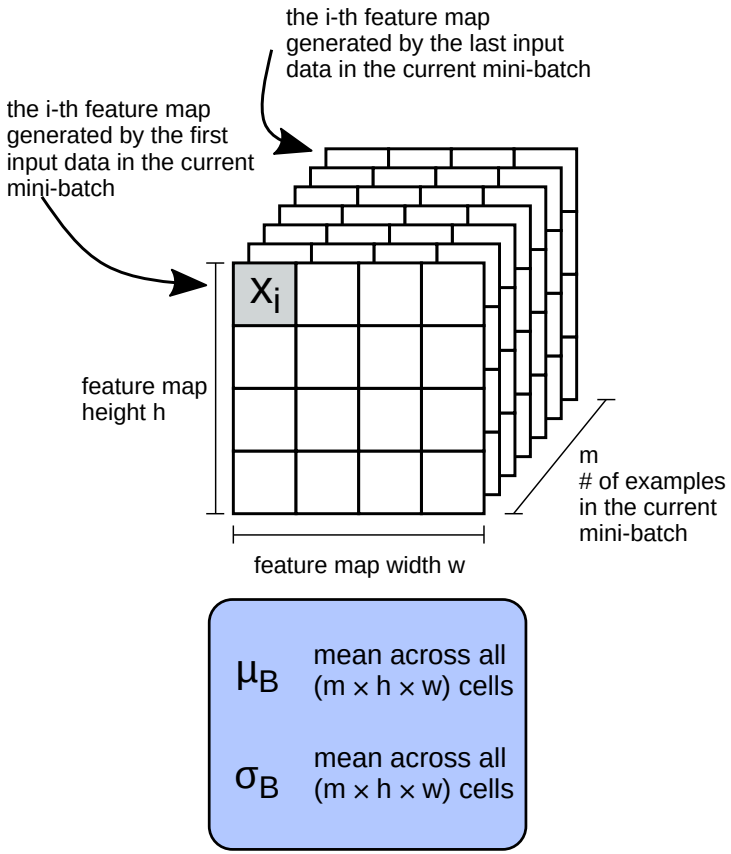
Early stopping is a cross-validation strategy where we keep one part of the training set as the development set. When the performance on the development set is getting worse, we immediately stop the training.



The idea of Early Stopping is to retrieve the parameter values (weights and biases) after stopping and using them as these are the values for which the model performs best.
Because the best fit is biased towards the development set, the model should then be evaluated on the test set to obtained an impartial result.

Regularization

Batch Normalization makes the activations Gaussian. This technique by Ioffe & Szegedy (2015) helps with improving the gradient flow through the network, allowing higher learning rates. Also, the strong dependence on parameter initialization is reduced.



input: values x of a mini-batch $B = \{x_1, ..., x_m\}$

output: $\{\hat{y}_i = BN_{\gamma,\beta}(x_i)\}$, parameters to be learned: γ, β

algorithm:

mini-batch mean $\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$

mini-batch variance $\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$

normalize $\tilde{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ where ϵ is added in the denominator for numerical stability and is an arbitrarily small constant

scale and shift $\hat{y}_i \leftarrow \gamma\tilde{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$

Regularization

Why does **Batch Normalization** work?

Regularization

Why does **Batch Normalization** work?

One benefit of batch normalization is that it acts as **regularization**: Each mini-batch is scaled using its mean and standard deviation, thus introducing some noise to each layer, providing a regularization effect.

Regularization

Why does **Batch Normalization** work?

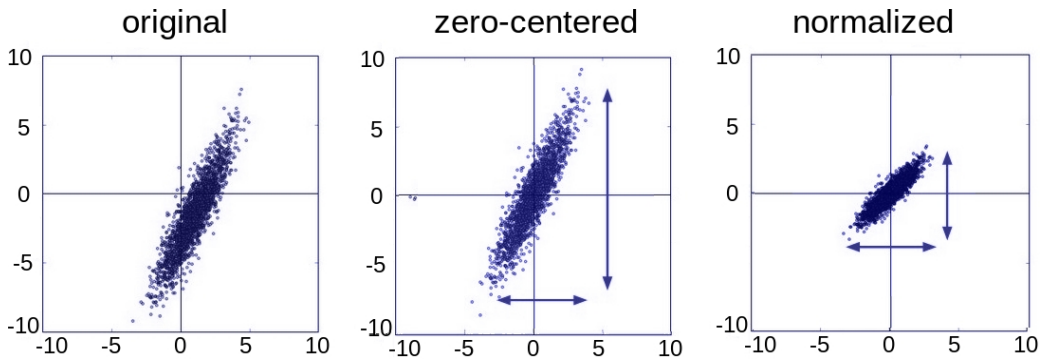
One benefit of batch normalization is that it acts as **regularization**: Each mini-batch is scaled using its mean and standard deviation, thus introducing some noise to each layer, providing a regularization effect.

The other benefit comes from the idea of **covariante shift**:

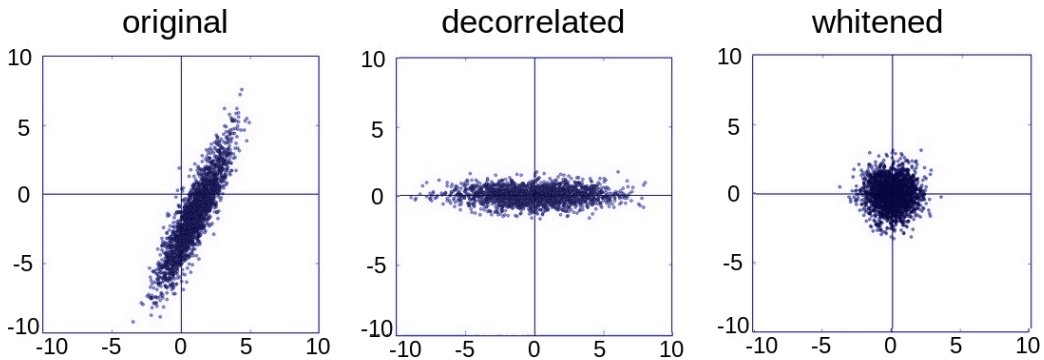
Suppose your training set for galaxy classification contains no highly redshifted galaxies.
The learned model will not perform well on realistic survey data from deep surveys. The reason is the shift in the input distribution, known as the covariate shift. Batch normalization is reducing this covariate shift.
The idea is that even when the exact values of inputs to hidden layers change, their mean and standard deviation will still almost remain same thus reducing the covariate shift. This weakens the coupling between parameters of early layer and that of later layers hence, allowing each layer of the network to learn by itself i.e. more independent of each other. This has the effect of speeding up the learning process.

Summary: The Complete Training Process

Step 1: Data Preprocessing



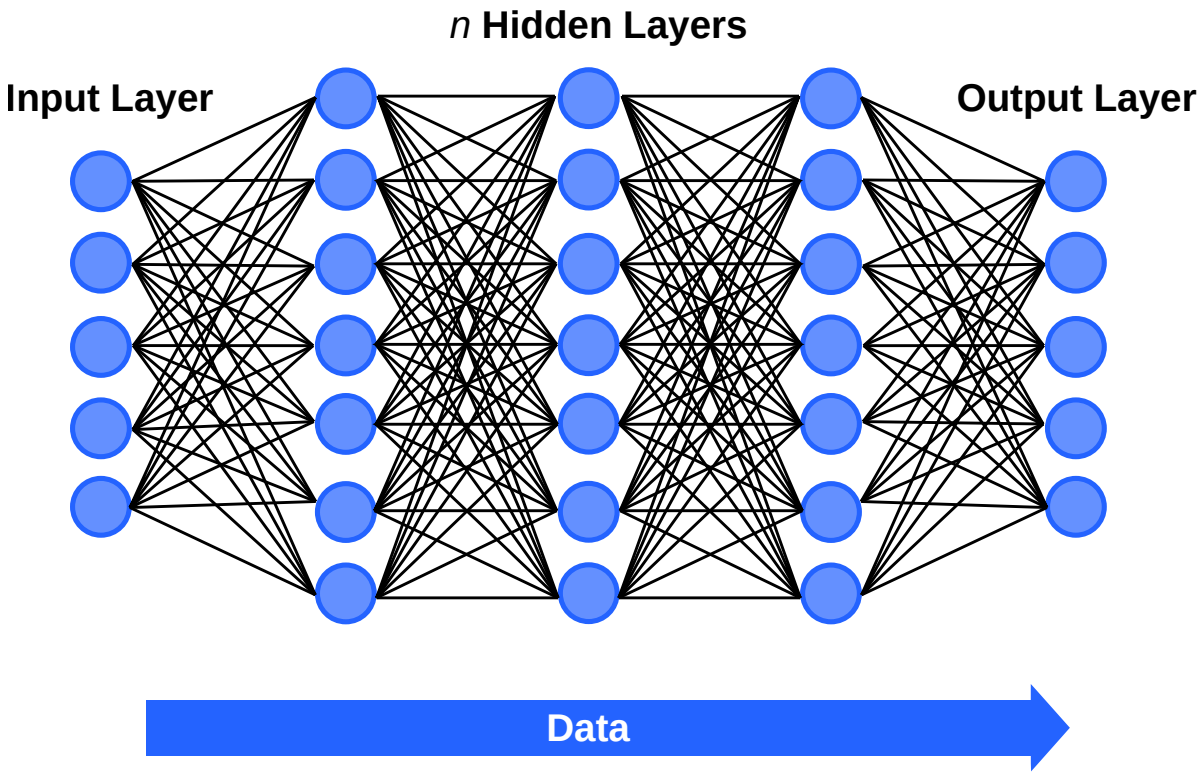
Common data preprocessing pipeline. Left: Original 2D input data.
Center: Zero-centering the data by subtracting the mean in each dimension.
Right: Additionally scaling by the dimension's standard deviation gives equal extends.



Data preprocessing pipeline with PCA / Whitening. Left: Original 2D input data.
Center: After performing PCA the data is centered at zero and decorrelated by being rotated into the eigenbasis of the data covariance matrix. Right: Additionally scaling by the eigenvalues stretches and squeezes the data into an isotropic Gaussian.

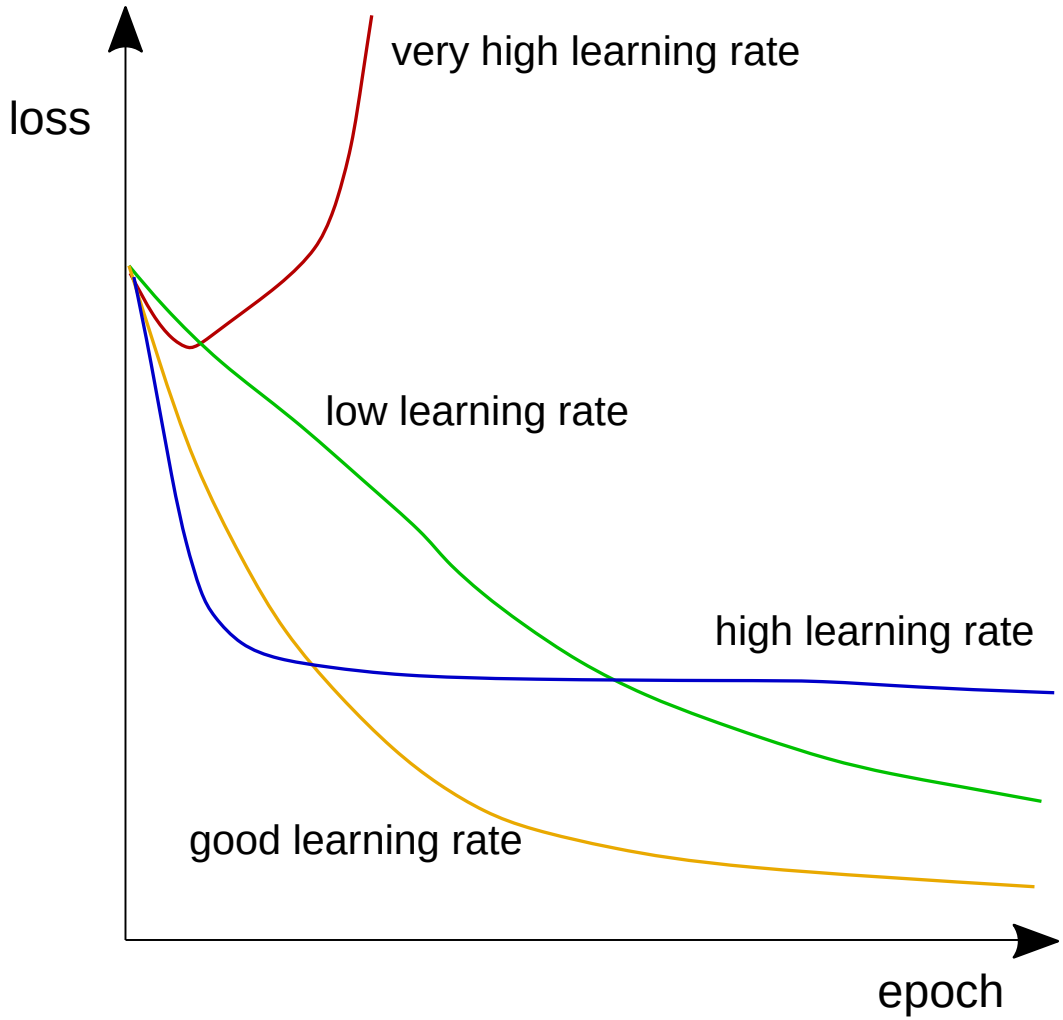
Summary: The Complete Training Process

Step 2: Chosing the Neural Network Architecture



Summary: The Complete Training Process

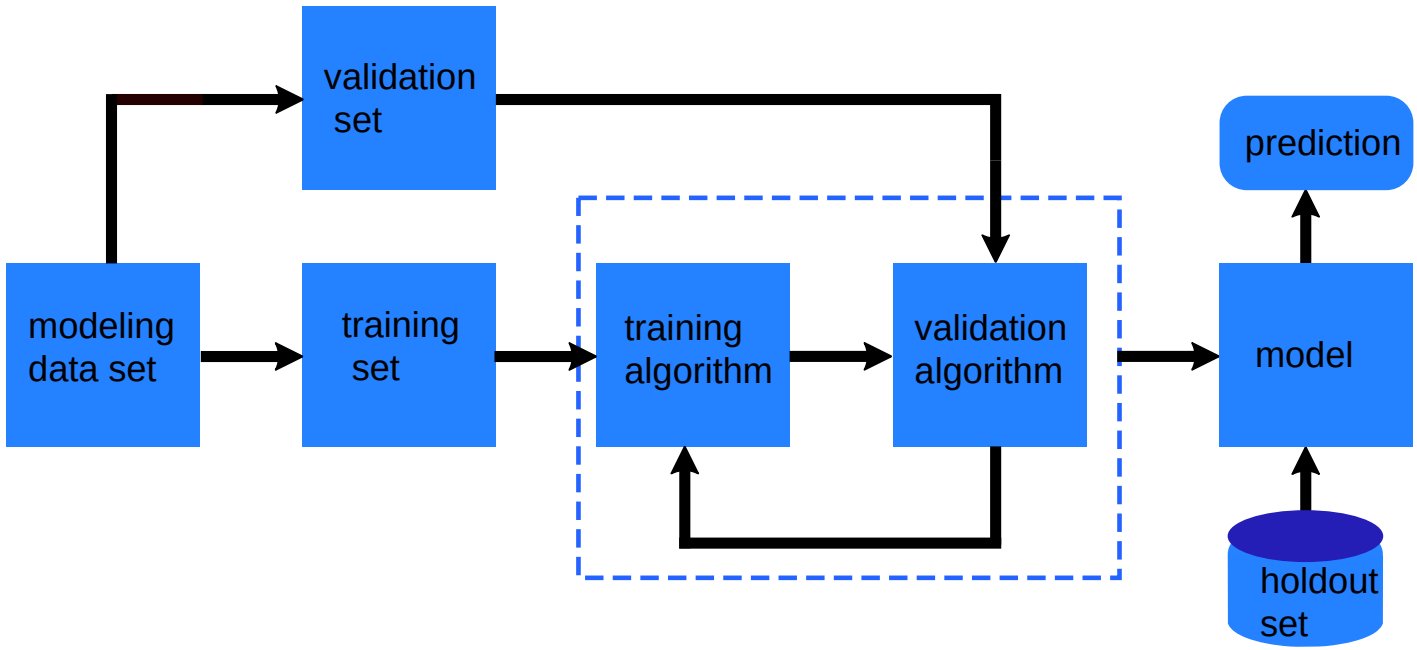
Step 3: Check the Loss



Verification

General rules for training of machine learning algorithms:

separate data used for training (the modeling set) into training, validation and holdout data sets

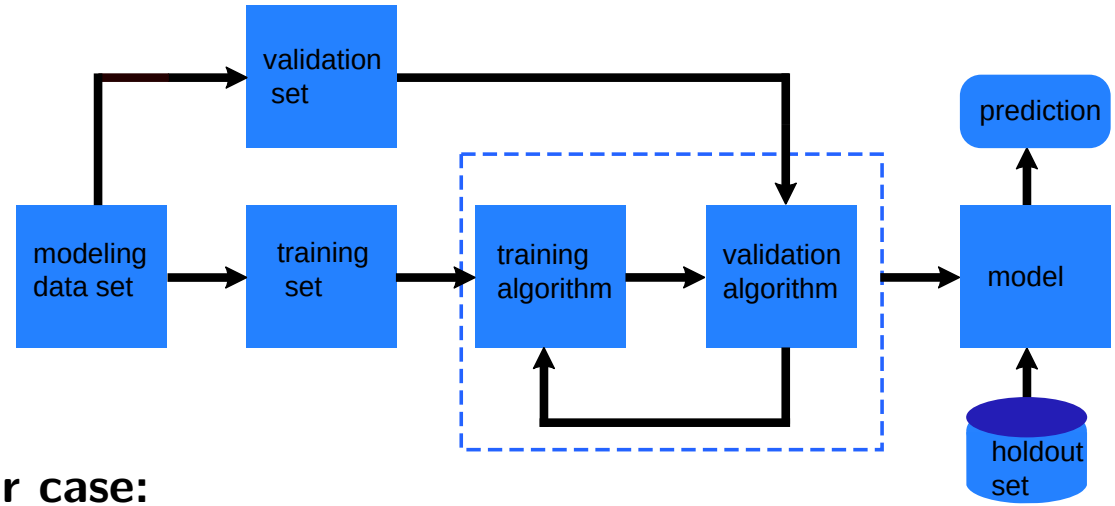


- **training data**: fit parameters of the network
- **validation (test) data**: evaluate the performance on unseen data
- **holdout data**: assess the final preformance of the tuned model

Verification

General rules for training of machine learning algorithms:

separate data used for training (the modeling set) into training, validation and holdout data sets



in our case:

- Training set determines the network parameters: weights and biases (to be evaluated).
- Training results validated on validation data: set network architecture (e.g.: early stopping).
- Use holdout data set to describe final performance.

Verification

For simplicity, we consider here binary classification where each observation is assigned to either class 1 or 0 (= not 1).
In that case, there are the following outcomes (if the goal is to identify class 1):

- True Positive = correctly identified (class 1 identified as class 1)
- True Negative = correctly rejected (class 0 rejected as class 0)
- False Positive = incorrectly identified (class 0 identified as class 1)
- False Negative = incorrectly rejected (class 1 rejected as class 0)

Verification

Based on these, we define either of the following pairs of terms:

$$\text{completeness} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

$$\text{contamination} = \frac{\text{false positives}}{\text{true positives} + \text{false positives}} = \text{false discovery rate}$$

Instead of contamination, often also efficiency (also called purity) is used:
$\text{efficiency} = (1 - \text{contamination})$

or

$$\text{true positive rate} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

$$\text{false positive rate} = \frac{\text{false positives}}{\text{true negatives} + \text{false positives}}$$

Similarly

$$\text{efficiency} = 1 - \text{contamination} = \text{precision}.$$

Verification

To illustrate the differences between these measures, let's look at the following **example:**

We have a modeling set containing 100,000 stars and 1000 quasars. If you correctly identify 900 quasars and mistake 1000 stars for quasars, we have:

Verification

To illustrate the differences between these measures, let's look at the following **example:**

We have a modeling set containing 100,000 stars and 1000 quasars. If you correctly identify 900 quasars and mistake 1000 stars for quasars, we have:

- TP = 900 (true positive)
- FN = 100 (false negative)
- TN = 99,000 (true negative)
- FP = 1000 (false positive)

Which gives

$$\text{true positive rate} = \frac{900}{900 + 100} = 0.9 = \text{completeness}$$

$$\text{false positive rate} = \frac{1000}{99000 + 1000} = 0.01$$

Verification

To illustrate the differences between these measures, let's look at the following **example:**

We have a modeling set containing 100,000 stars and 1000 quasars. If you correctly identify 900 quasars and mistake 1000 stars for quasars, we have:

- TP $= 900$ (true positive)
- FN $= 100$ (false negative)
- TN $= 99{,}000$ (true negative)
- FP $= 1000$ (false positive)

Which gives

$$\text{true positive rate} = \frac{900}{900 + 100} = 0.9 = \text{completeness}$$

$$\text{false positive rate} = \frac{1000}{99000 + 1000} = 0.01$$

Despite the FPR doesn't look bad, there are a lot of stars, so the contamination rate isn't good: $\text{contamination} = \frac{1000}{900+1000} = 0.53$

Verification

To illustrate the differences between these measures, let's look at the following **example:**

We have a modeling set containing 100,000 stars and 1000 quasars. If you correctly identify 900 quasars and mistake 1000 stars for quasars, we have:

- TP $= 900$ (true positive)
- FN $= 100$ (false negative)
- TN $= 99{,}000$ (true negative)
- FP $= 1000$ (false positive)

Which gives

$$\text{true positive rate} = \frac{900}{900 + 100} = 0.9 = \text{completeness}$$

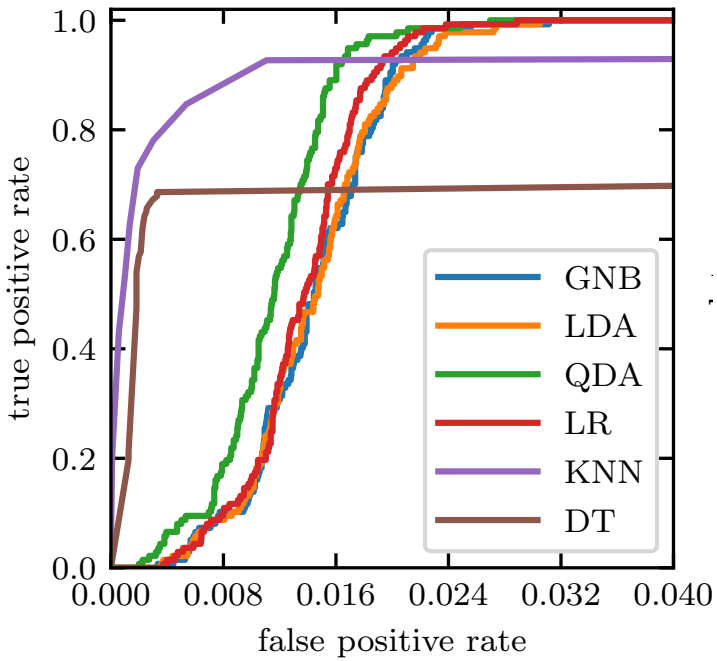
$$\text{false positive rate} = \frac{1000}{99000 + 1000} = 0.01$$

however:
The classifier might be sufficient as one step in a classification pipeline.

Verification

calculate these performance measures by applying **10-fold cross-validation**: in turn a random 10 % held out ⇒ train on 90%, apply to 10%

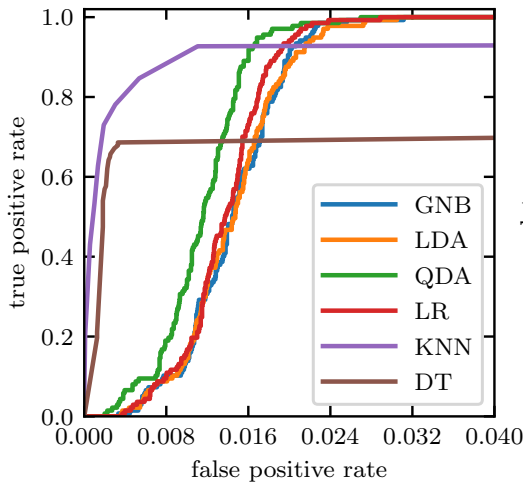
⇨ from this we quantify the tradeoff using a **Receiver Operating Characteristic (ROC) curve** which plots the true-positive vs. the false-positive rate



Verification

calculate these performance measures by applying **10-fold cross-validation**: in turn a random 10 % held out ⇒ train on 90%, apply to 10%

⇒ from this we quantify the tradeoff using a **Receiver Operating Characteristic (ROC) curve** which plots the true-positive vs. the false-positive rate

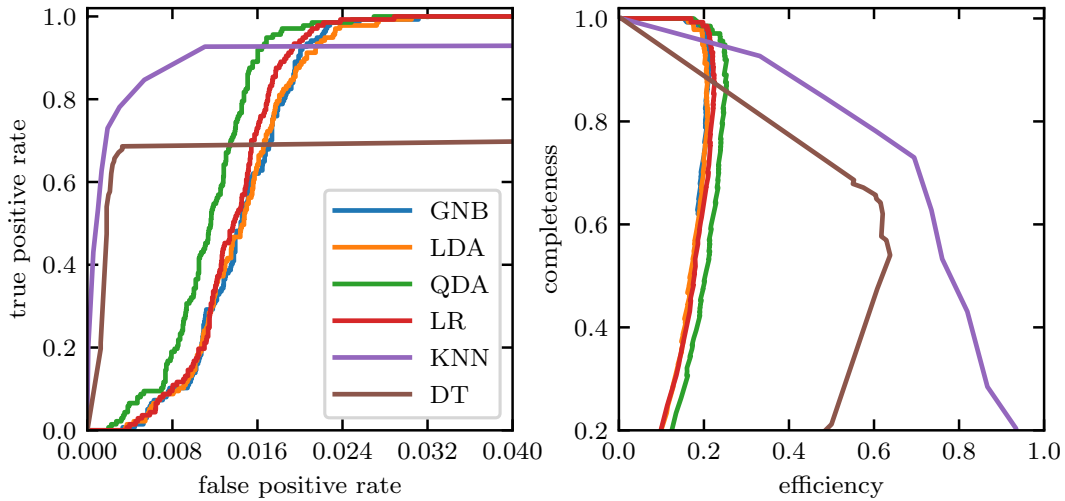


concern: ROC curves are sensitive to the relative sample sizes: in the case of # background events ≫ # source events, small false positive results can dominate a signal.

solution: For these cases we plot completeness versus efficiency.

Classifier Performance

Here is a comparison of the two types of plots:



We see that in order to get higher completeness, you could actually suffer significantly in terms of efficiency, but the FPR might not go up that much if there are lots of true negatives.

Note that the desired completeness and efficiency is chosen by selecting a **decision boundary**. The curves show what these possible choices are. Generally, one wants to chose a decision boundary that maximizes the area under the ROC (or completeness versus efficiency) curve.

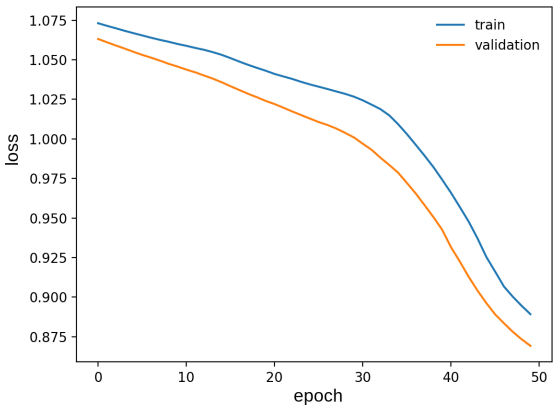
Neural Network-Specific Performance Metrics

Learning Curves:

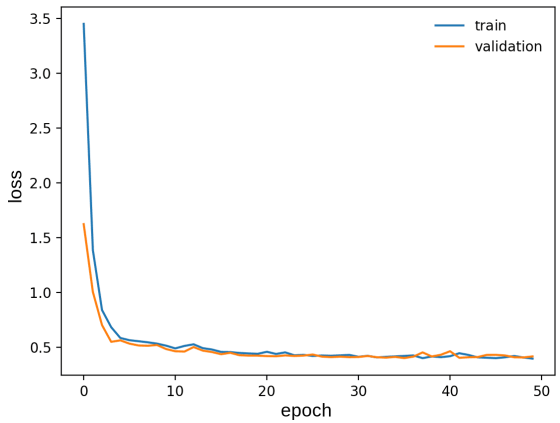
Plot loss vs. number of training steps for both training set and validation set

Ideal case: both training and validation set learning curves converge to a low number

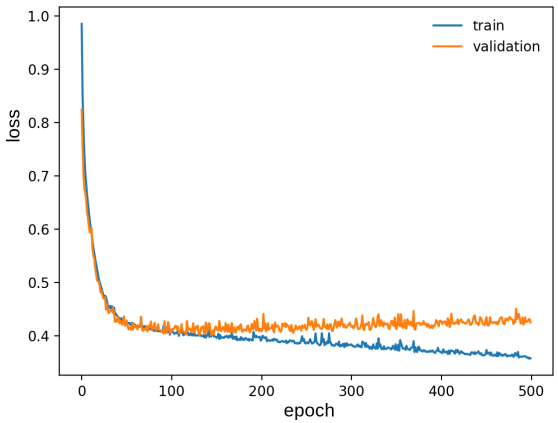
If the curve for the validation set starts to go up, this indicates overfitting



underfitting - more training necessary



good fit



overfitting

Workflow for Developing Neural Network Applications

1. understand and define the problem:
   - what are the inputs?
   - what are the expected outputs? (classification (binary, multiclass, multilabel); regression)
   - are inputs sufficently informative to determine outputs?
2. define the data set
   - prepare data: normalization, PCA
3. choose a metric to measure success
4. Define evaluation methodology
   - e.g. cross-validation: holdout if plenty of data, otherwise K-fold
5. develop mode
   - last-layer activation function
   - loss function
   - optimization
6. Train model to point of overfitting
7. Tune the model: regularization
8. Test the model
9. If sufficient: apply the model

Software Frameworks

TensorFlow

https://www.tensorflow.org/

created by Google

used for machine learning and especially neural networks
supports Python and C++

⊕ documentation
large community, many tutorials
built-in monitoring for training processes (Tensorboard)

⊖ static computational graphs for neural network models
some performance issues

Software Frameworks

https://www.pytorch.org/

created by Facebook

tensor computation (like numpy) with GPU acceleration
supports Python

⊕ dynamic computational graphs (useful for e.g. RNN*)
easy to write own layer types

⊖ lacks built-in monitoring
documentation not as good

* we will see this later on

Software Frameworks

Keras

https://www.keras.io/

Keras is built on top of TensorFlow/ Theano
supports Python

⊕ very intuitive interface for building neural networks
easy to write own layer types

⊖ not as many functionalities, less control

Software Frameworks

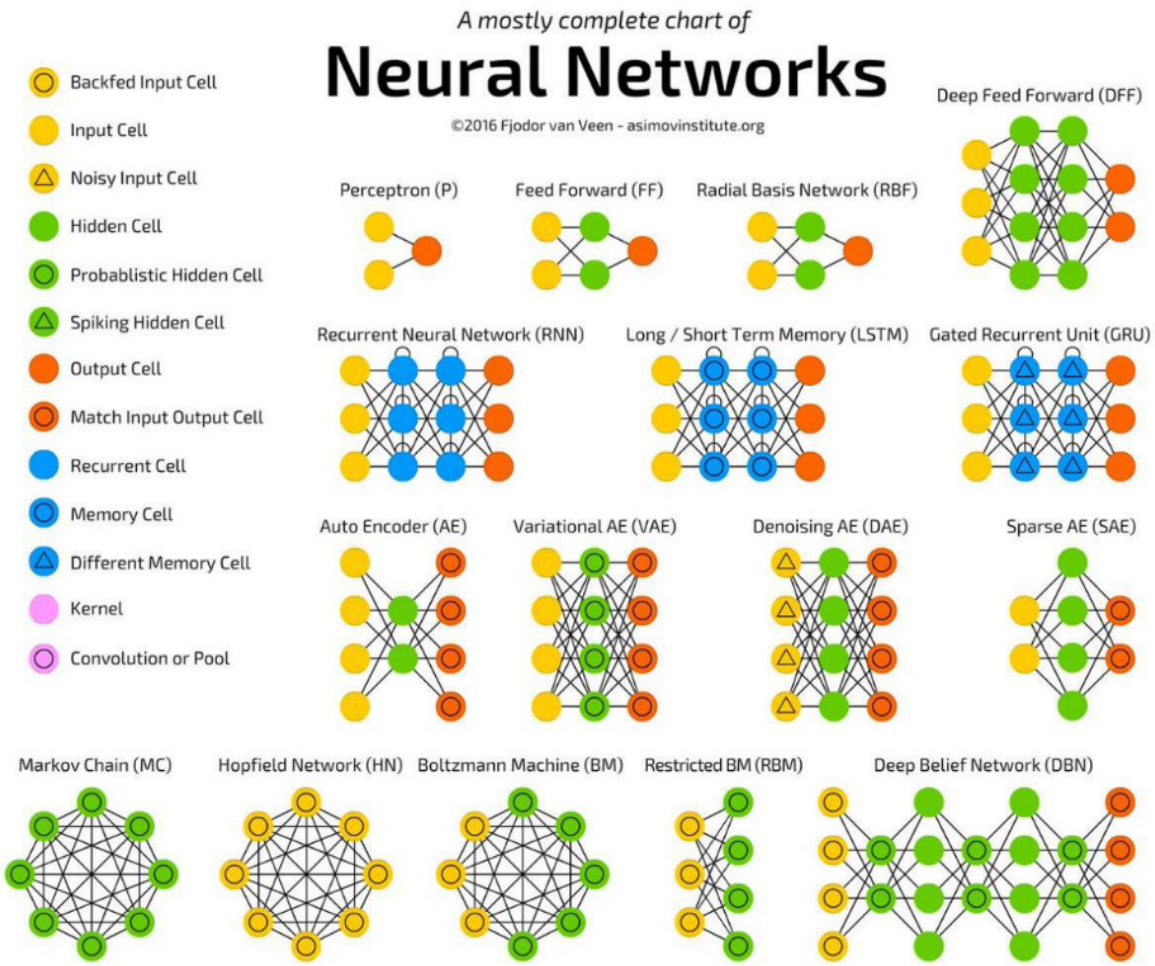
https://astronn.readthedocs.io/en/latest/

astroNN is a neural ntwork package especiall for **astronomy**.

astroNN uses Keras API for model and training prototyping, but at the same time take advantage of Tensorflow's flexibility.

Whereas originally designed to apply neural networks on APOGEE spectra, tools are included to deal with APOGEE, Gaia and LAMOST data.

An Outlook: Neural Network Architectures



Sasen Cain (@spectralradius)