

Intro to deep learning 1

26. november 2024 14:44

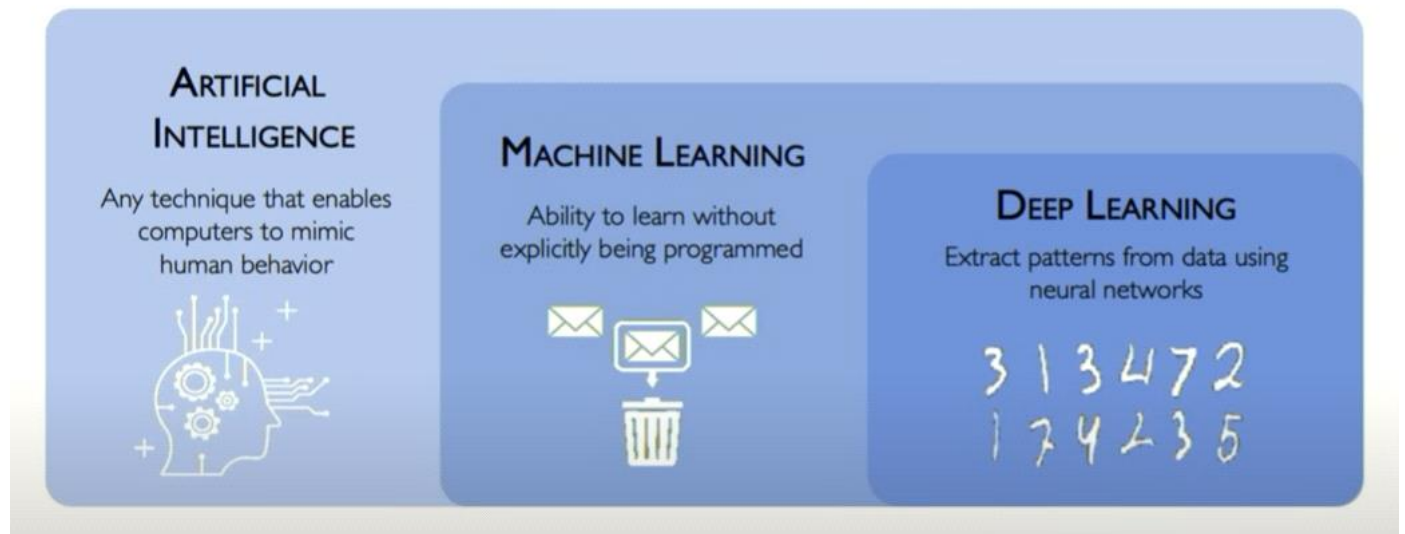
Lecture 1

Artificial intelligence. Process information to influence future decisions.

ML: How to do that with data without encoding specific rules.

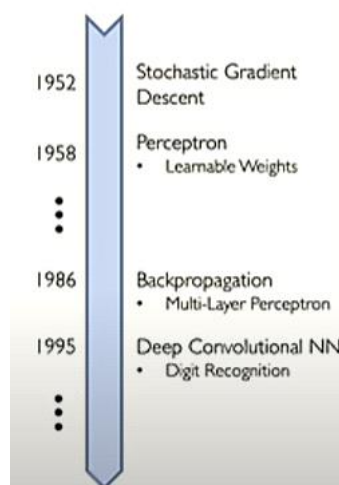
DL: Extract patterns using neural networks

What is Deep Learning?

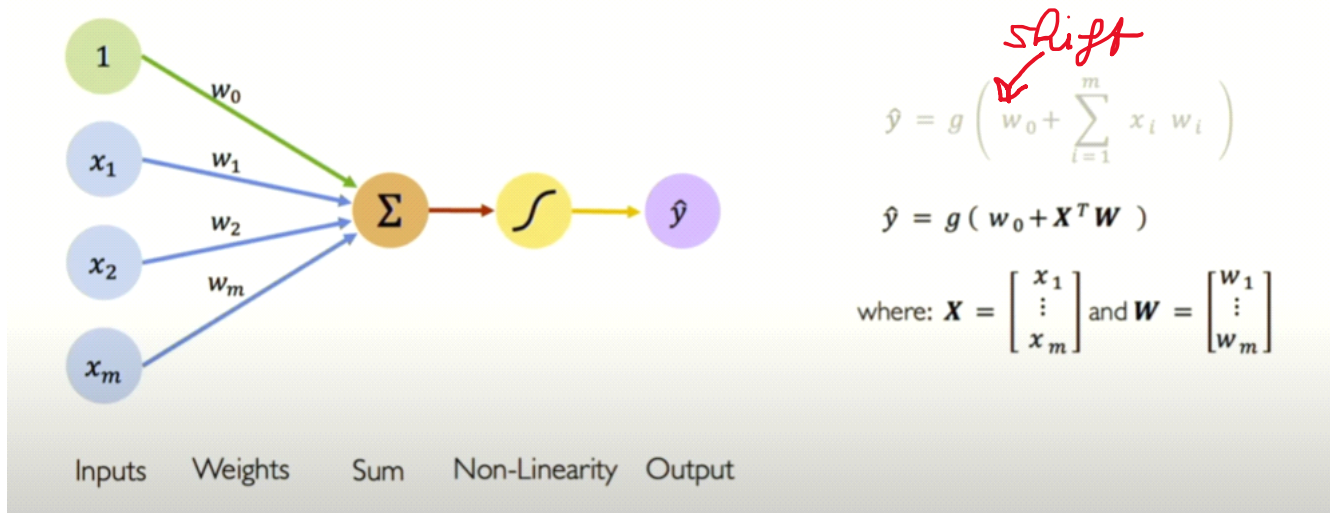


Why DL? Usually features are hand engineered for the data, but hardly scalable, so we try to move away from hand made rules and features, and we try to learn them from RAW pieces of data

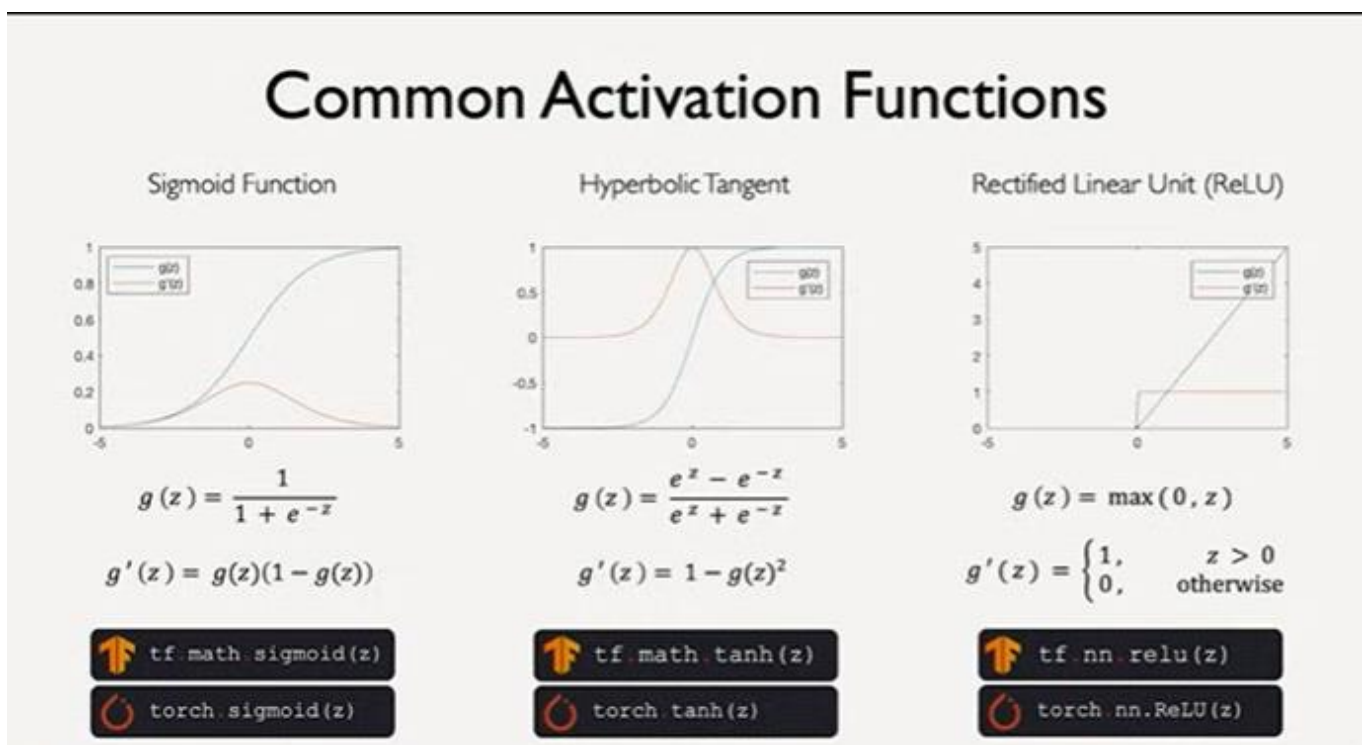
Building elements of NN are old and known, but enormous amounts of data, hardware and usable software solutions have made DL widespread and possible in practice



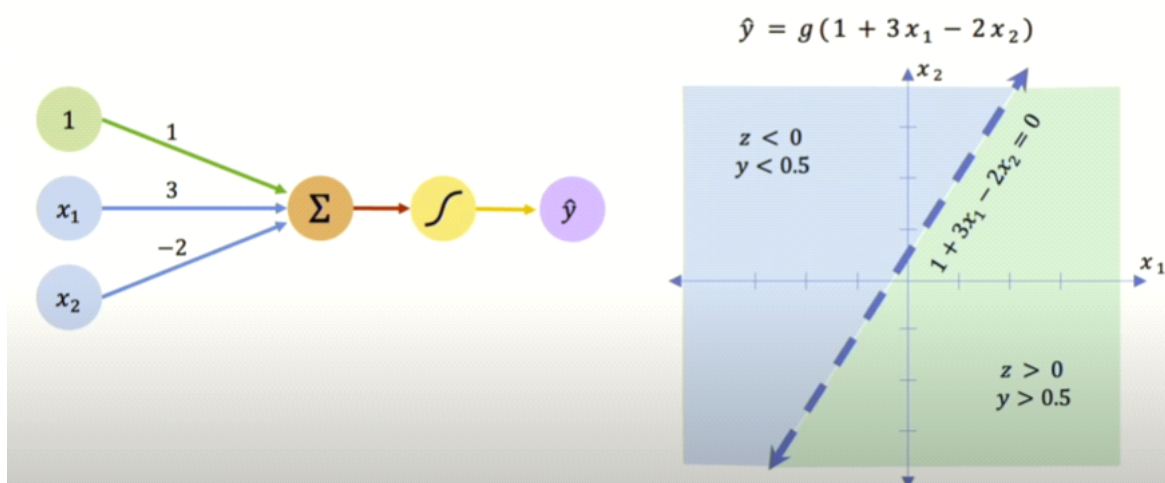
Perceptron - basic element of the NN, a single neuron



After the summation node of dot product and bias, the result enters a non-linear function g (yellow node). A choice can be sigmoid, popular because it has output in (0,1). There are others, like tanh and relu.



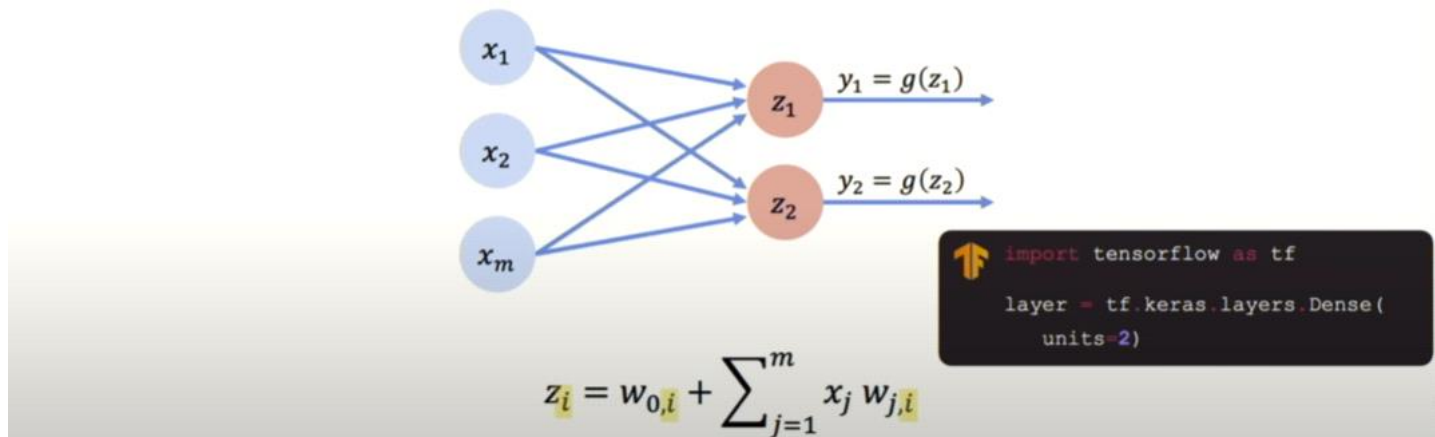
g allows the network to deal with **non-linearity in the data**. The perceptron divides the space in two and then non-linearizes any point using the dot product function inside $g()$.



You can use multiple perceptrons. Here the input is a dense layer, because all inputs go to all neurons. z is the dot product now. Those kind of layers are all implemented in one line

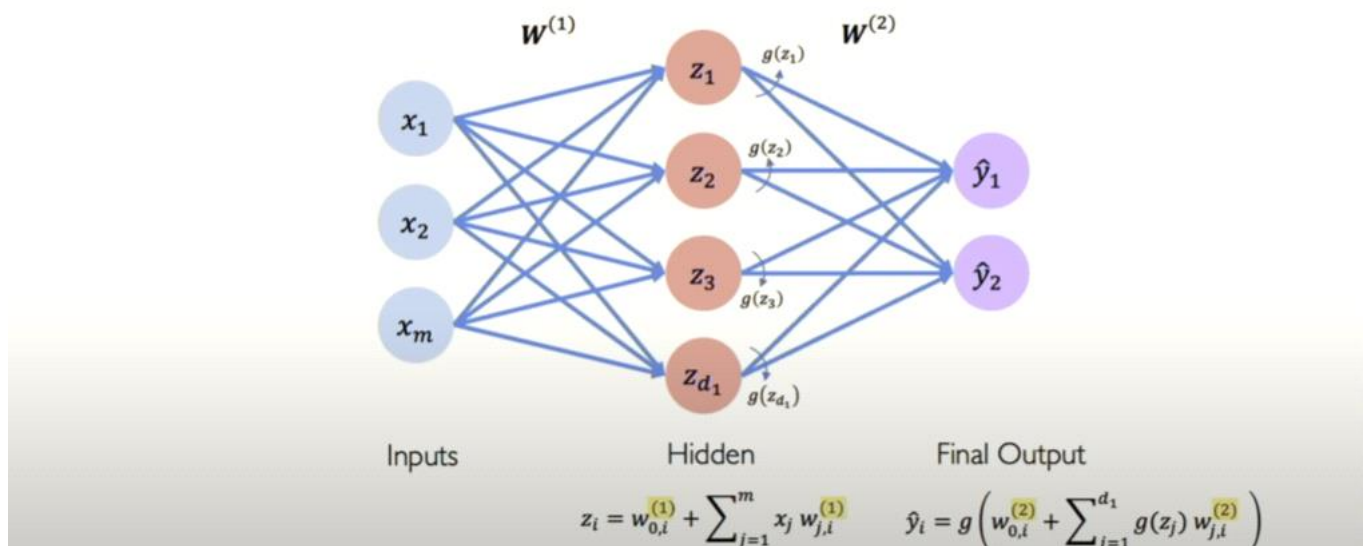
Multi Output Perceptron

Because all inputs are densely connected to all outputs, these layers are called **Dense** layers



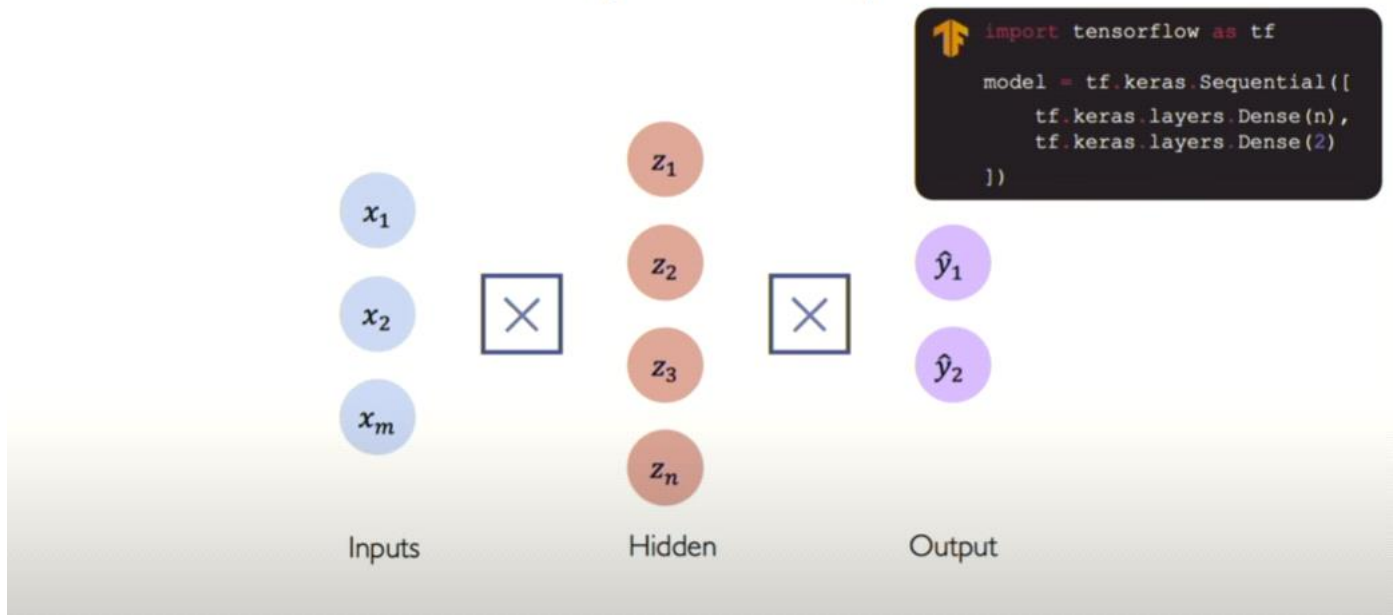
By extension, a single layer NN has the neurons as a hidden layer, whose non linear outputs combine densely using a second set of weights $W^{(2)}$. Note that every layer needs its own non linear function. The Y outputs are done using the g function, which in turns take the perceptrons output transformed by g , so you have a cascade of non linear functions. Otherwise, you would have multiple layers which are all linear and do not add anything to the network.

Single Layer Neural Network



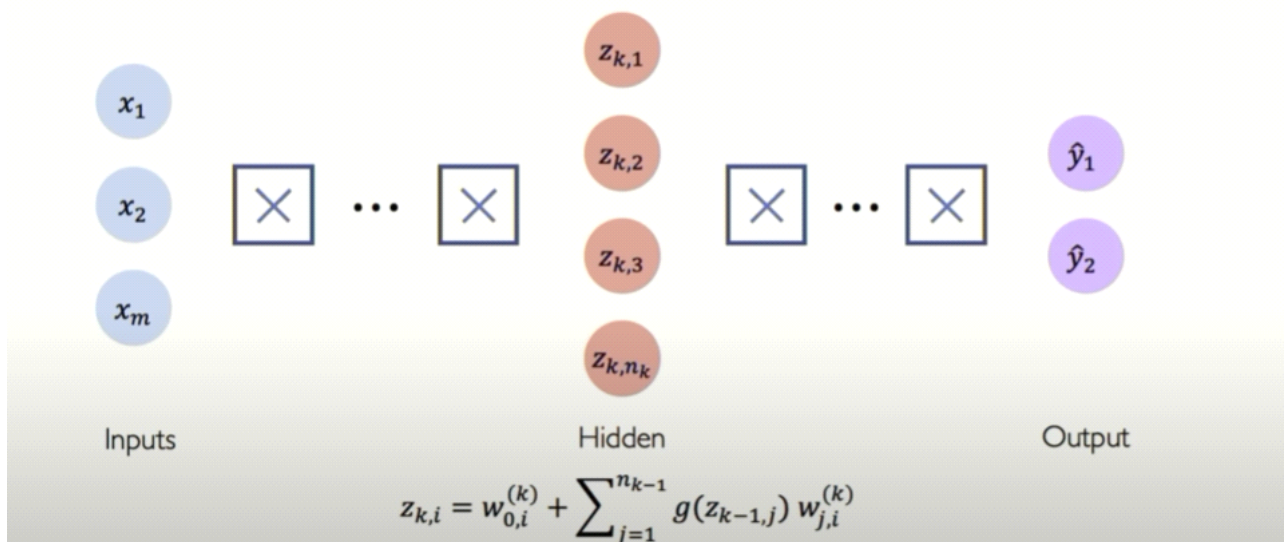
Again, everything is coded with keras

Multi Output Perceptron



A deep NN has a stack of neuron layers, and the formula is simply recurrent depending on the previous layer

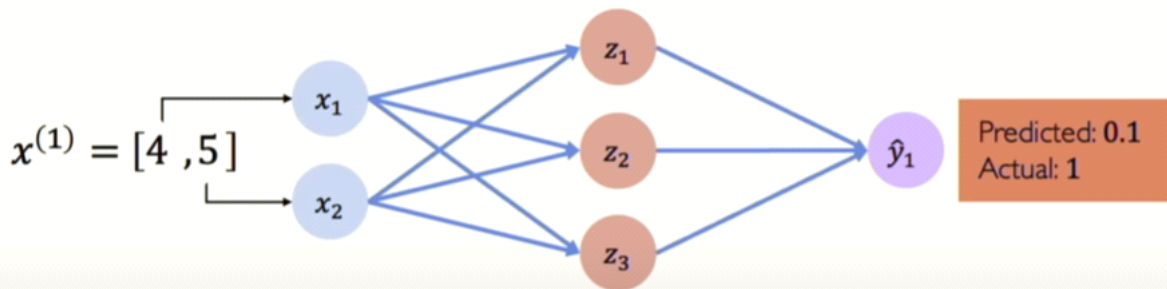
Deep Neural Network



A NN needs to be trained, so that it can predict the output from a new input. This is done with a loss function, where you compare predictions on training data with true values to predict. The weights need to be changed to reduce the loss function. The example here is for data with two categories.

Quantifying Loss

The **loss** of our network measures the cost incurred from incorrect predictions

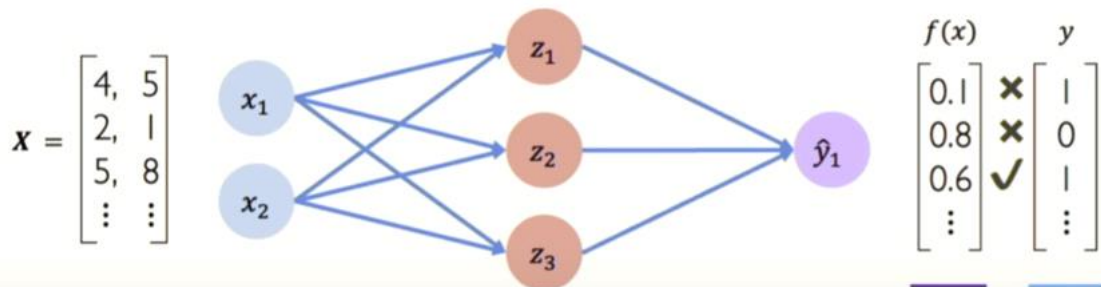


$$\mathcal{L}(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

In problems like this the cross entropy function can be a choice of loss. This determines how far away from each other the predicted and true probability distributions are.

Binary Cross Entropy Loss

Cross entropy loss can be used with models that output a probability between 0 and 1



$$J(\mathbf{W}) = -\frac{1}{n} \sum_{i=1}^n \underbrace{y^{(i)}}_{\text{Actual}} \log(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}}) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log(1 - \underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}})$$

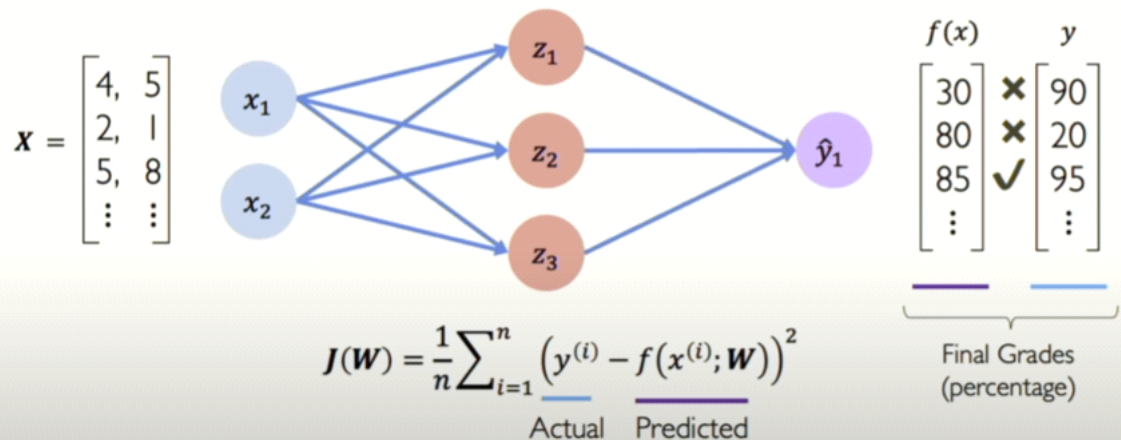


```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```

For continuous values you can use other loss functions like MSE

Mean Squared Error Loss

Mean squared error loss can be used with regression models that output continuous real numbers



```
loss = tf.reduce_mean( tf.square(tf.subtract(y, predicted)) )
loss = tf.keras.losses.MSE( y, predicted )
```

Again, standard losses can all be found in keras!

Optimization of $J(W)$ to find all optimal weights are done with gradient descent. Eta gives the amount of descent/optimization done at every step.

Loss Optimization

We want to find the network weights that *achieve the lowest loss*

$$W^* = \underset{W}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

$$W^* = \underset{W}{\operatorname{argmin}} J(W)$$

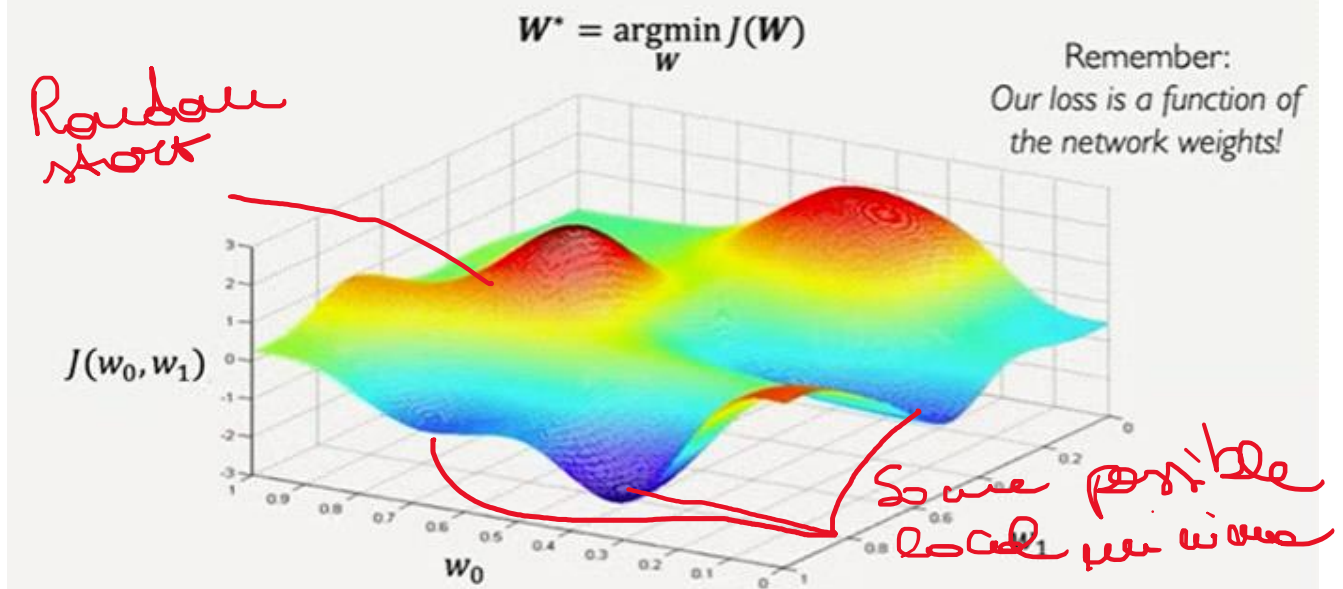
Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$
4. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

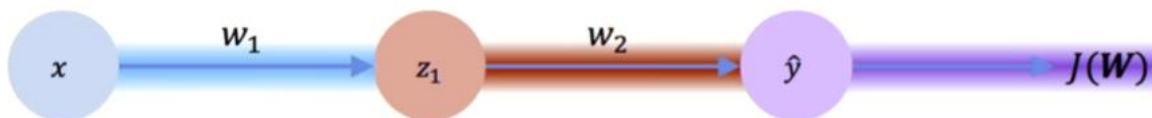
Step size

Loss Optimization



The gradient wrt each weight is done with the backpropagation! It uses the chain rule

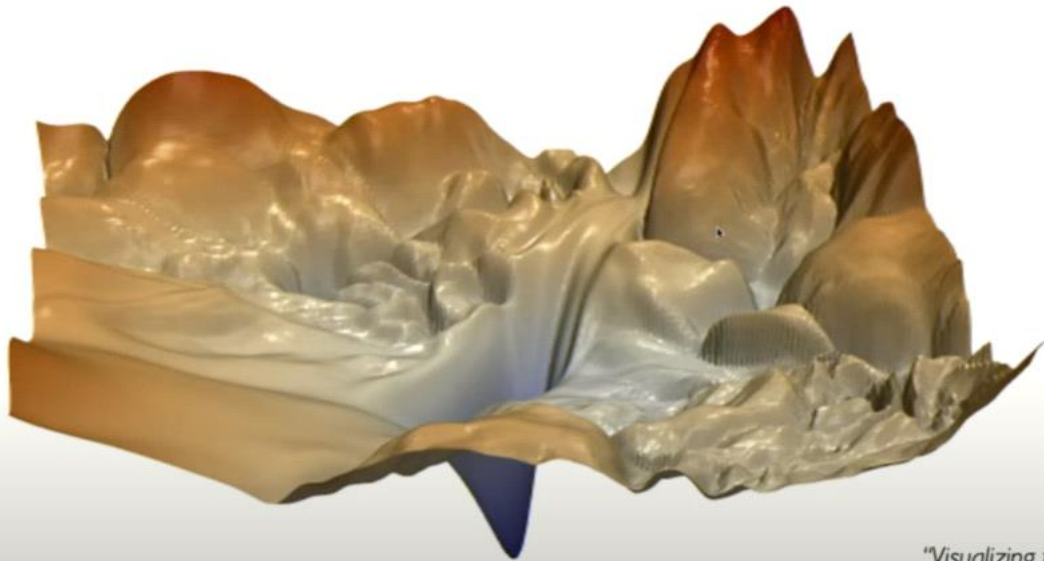
Computing Gradients: Backpropagation



$$\frac{\partial J(W)}{\partial w_1} = \underbrace{\frac{\partial J(W)}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}}_{\text{red}} * \underbrace{\frac{\partial z_1}{\partial w_1}}_{\text{blue}}$$

The loss landscape is very messy and therefore it can be hard to get into global minimum. One thing to do is to build NNs with more treatable loss landscapes, the other is to traverse the local minima to help finding the optimum (for example by changing the learning parameter eta in the gradient descent). Changing eta can help jumping more and avoid getting stuck in local minima.

Training Neural Networks is Difficult



"Visualizing the loss landscape of neural nets". Dec 2017.



There are algorithms adapting to the data (adaptive algorithm), the landscape and the gradient, by changing the learning rate during the optimization

Gradient Descent Algorithms

Algorithm

- SGD
- Adam
- Adadelata
- Adagrad
- RMSProp

TF Implementation

 <code>tf.keras.optimizers.SGD</code>
 <code>tf.keras.optimizers.Adam</code>
 <code>tf.keras.optimizers.Adadelata</code>
 <code>tf.keras.optimizers.Adagrad</code>
 <code>tf.keras.optimizers.RMSProp</code>

Reference

Kiefer & Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function." 1952.

Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.

Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.

Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.

Additional details: <http://ruder.io/optimizing-gradient-descent/>

Mini batches:

In gradient descent, the gradient calculation is very heavy, done at every step and over all data points. This is very expensive and basically unfeasible. This is why you use SGD, stochastic gradient descent, which only uses a batch of the data (mini batch, for example 32 data points is a common choice). You average the gradient over those samples - the gradient is noisy but now very feasible to calculate. You can train much faster and mini batches can also be parallelized: you can split up a batch and parallelize the calculation of the gradient.

Overfitting: common in all ML

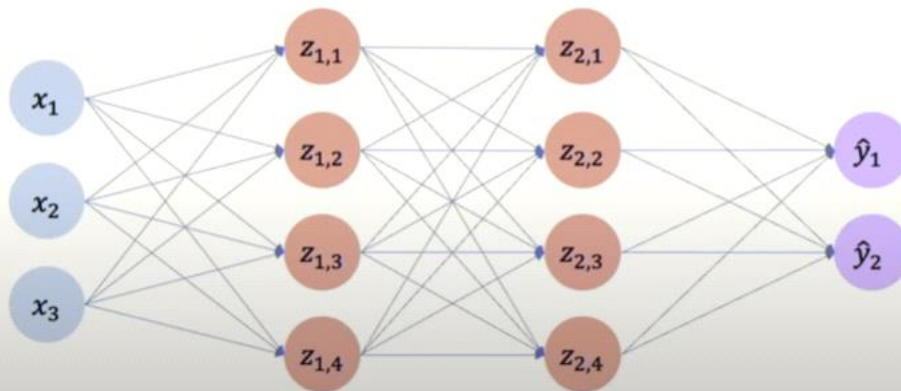
How can you define if your model is learning the true origin-manifold of the data without actually learning only the small details of the data you have. You want the trained model to generalize to other datasets originating from the same surface of your own data. Usually overfitting a model gives bad test data performance. Underfitting is the opposite with bad performance on test and training data.

One solution is REGULARIZATION, which penalizes models that are too complex, so that there is a balance between good fit and complexity.

DROPOUT regularization: some outputs of neurons are set to 0 with some probability. You change the dropout at every iteration, so that small part of the features of the data are not used all the time to inform the network during training

Regularization I: Dropout

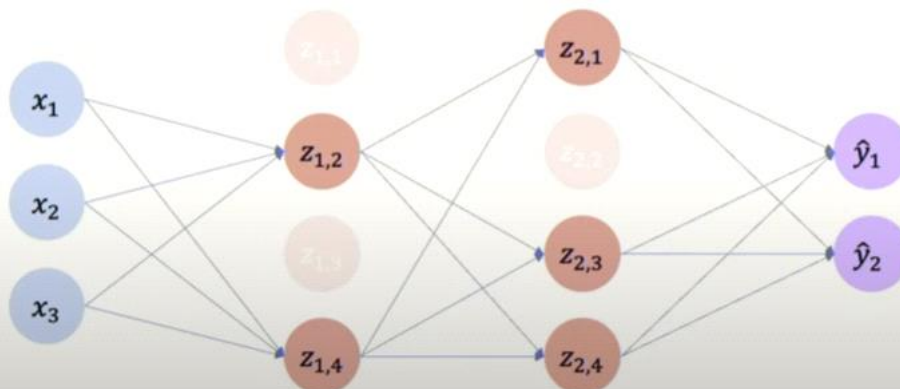
- During training, randomly set some activations to 0



Regularization I: Dropout

- During training, randomly set some activations to 0
 - Typically 'drop' 50% of activations in layer
 - Forces network to not rely on any 1 node

 `tf.keras.layers.Dropout(p=0.5)`



EARLY STOPPING regularization: stop training when testing accuracy becomes worse but the training becomes better

Regularization 2: Early Stopping

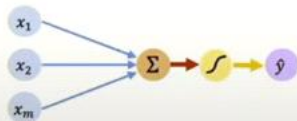
- Stop training before we have a chance to overfit



Core Foundation Review

The Perceptron

- Structural building blocks
- Nonlinear activation functions



Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization

