VICTORIA UNIVERSITY OF
WELLINGTON
TE HERENGA WAKA

AIML231/DATA302 —Techniques in Machine Learning

Week 8 Neural Networks (1)

Training Neural Networks

Dr Qi Chen

School of Engineering and Computer Science

Victoria University of Wellington

Qi.Chen@vuw.ac.nz

Outline

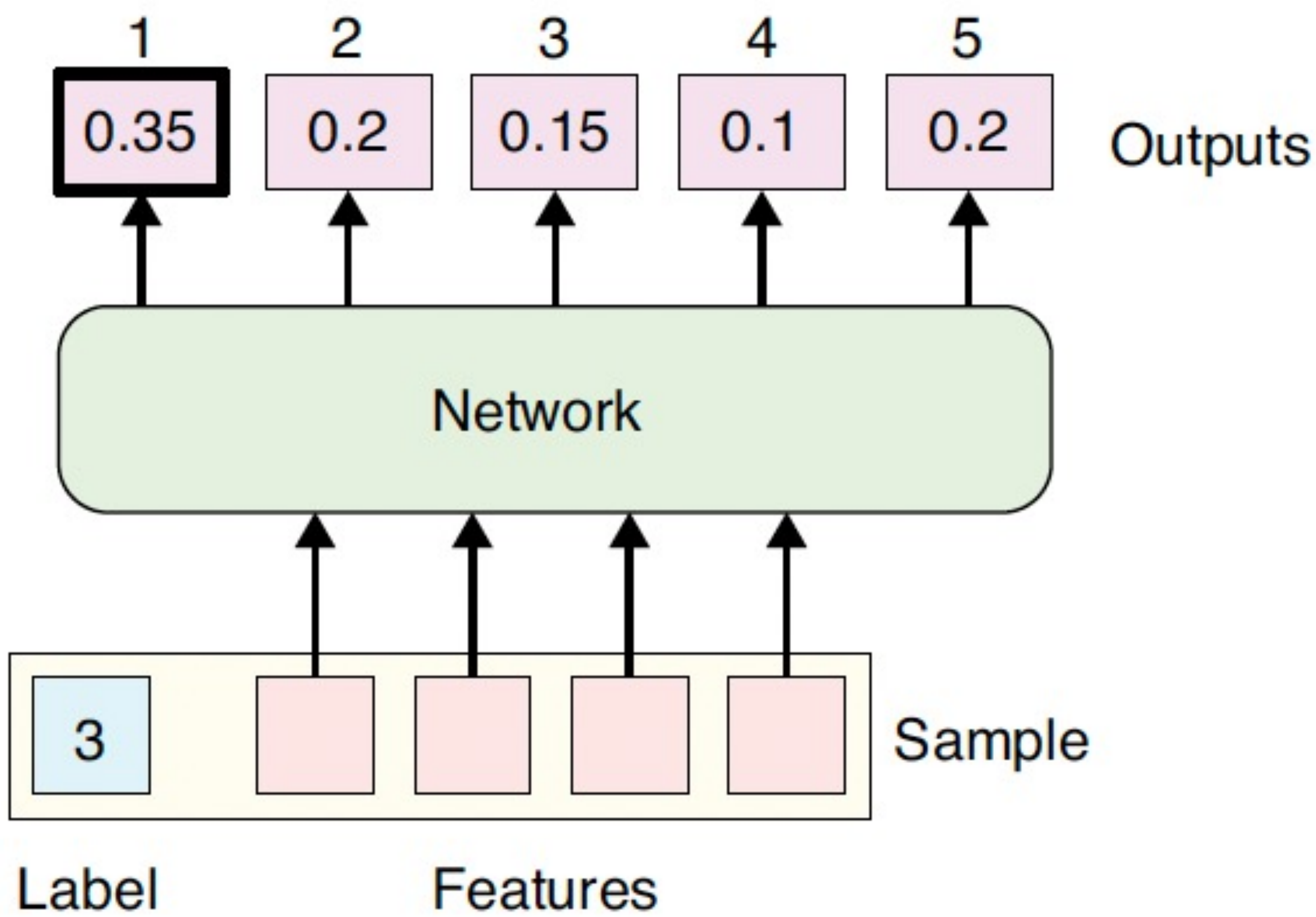
Training NN

- High Level Overview
- Loss function
- Optimization algorithms: Gradient Descent, Backpropagation
- Gradient variants
- Regularisation techniques

- Tutorial: Pytorch on NN Implementation
  - deliver by Dr. Junhong Zhao

Before Training NN

An example of NN Prediction without training



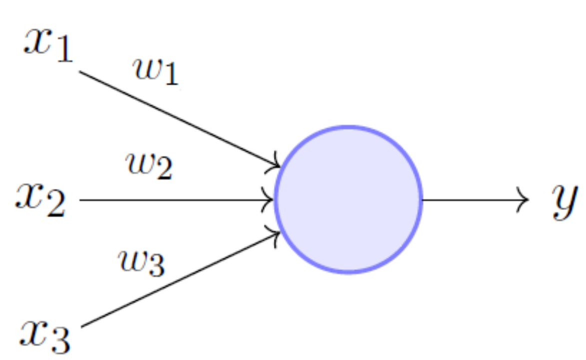
- *A neural network processing a sample and assigning it to class 1*

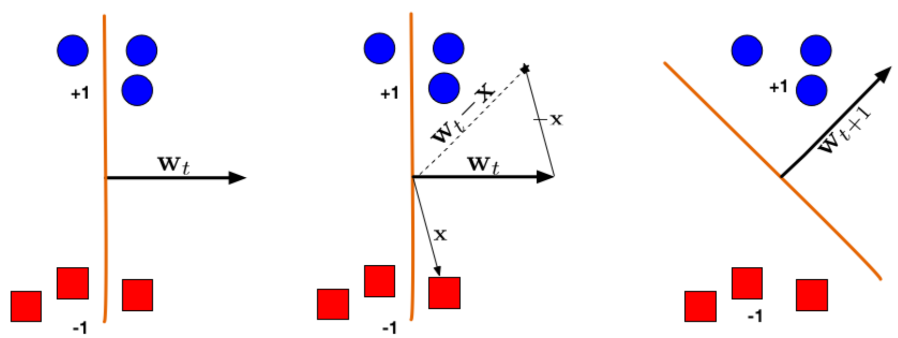
Perceptron Learning

– Initialise weights and threshold randomly (or all zeros)

– Given a new example $x_1, x_2, \ldots, x_p, d$

- Input feature vector: $x_1, x_2, \ldots, x_p$
- Output (class label): d
- Predicted (by perceptron) output





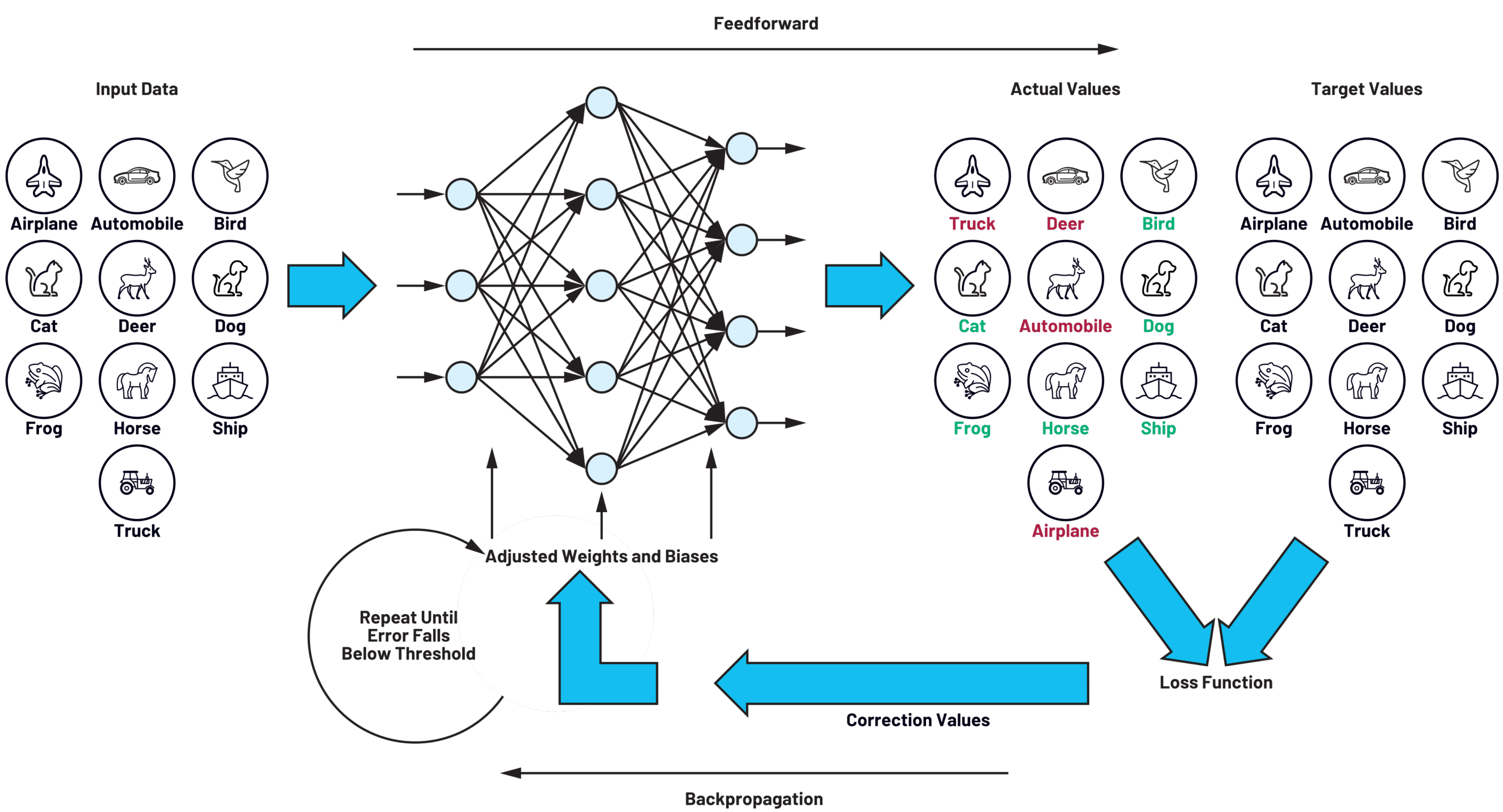
Basic learning algorithm:

- If $y = 0$ and $d = 1$:
  - increase w_i for positive x_i, decrease w_i for negative x_i
- If $y = 1$ and $d = 0$:
  - decrease w_i for positive x_i, increase w_i for negative x_i
- Repeat for each new example until achieve the desired behaviour
- Can also repeat all data and start again (multiple epochs)

Overview-Training a NN

Intuitively: teach a neural network to learn from mistakes thus making correct predictions

- adjusting weights either up or down so the error is reduced



https://www.analog.com/en/resources/analog-dialogue/articles/training-convolutional-neural-networks-what-is-machine-learning-part-2.html

Training a NN

- A process to find the optimal set of weights W^* that results in the smallest cumulative loss across all the training data

$$W^* = \underset{W}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, \widehat{y_i})$$

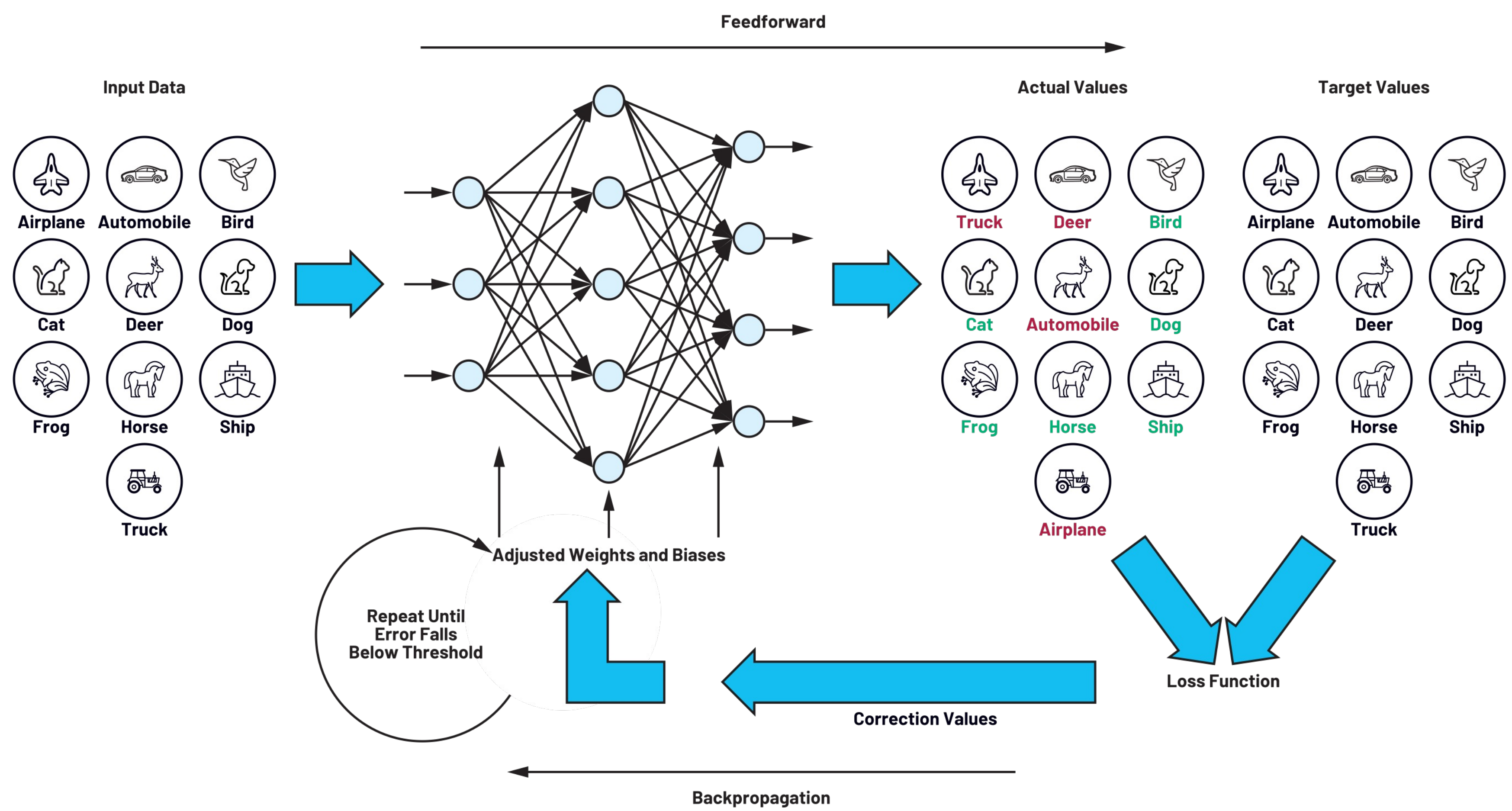
  - W represents the weights of the neural network
  - *L* represents the loss/cost/error function, measures the difference between the predicted output $\hat{y}_i$ and the actual target y_i
  - *N* is the total number of training samples

- A complex optimisation problem: often involving high dimensional search space, non-linear transformations…

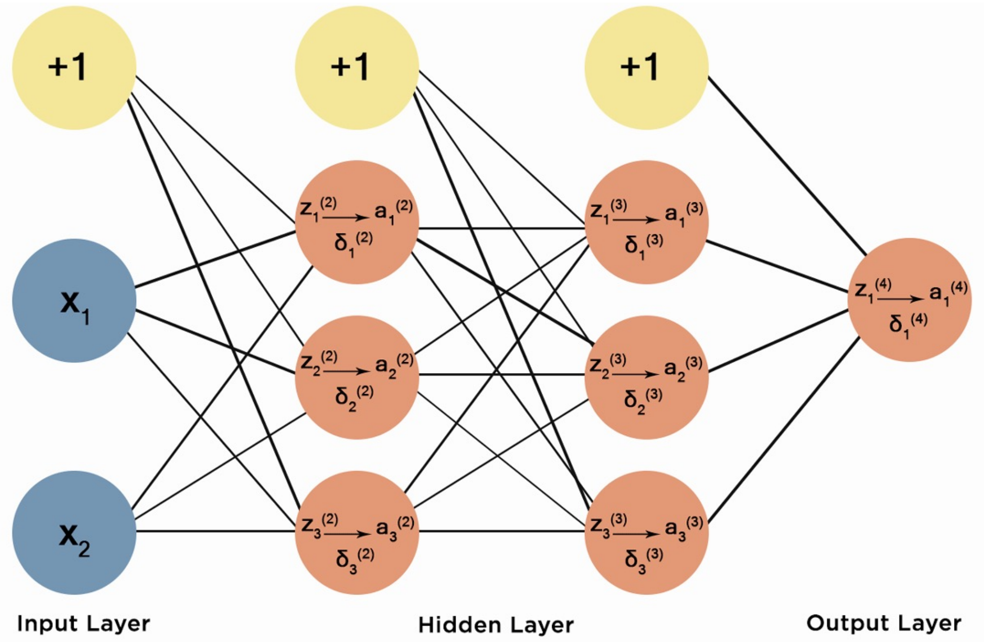
Training Loop

1. Initialisation: set up neural network with initial settings
2. Input Feeding
3. Making Prediction (Forward Pass): give the network a piece of data to look at, it attempts to predict the correct answer
4. Calculating Loss
5. Updating Parameters
6. Evaluate on a validation set (optional) – hyperparameters tuning

Optimisation step - often with gradient descent



Forward Pass: What does Network Compute?



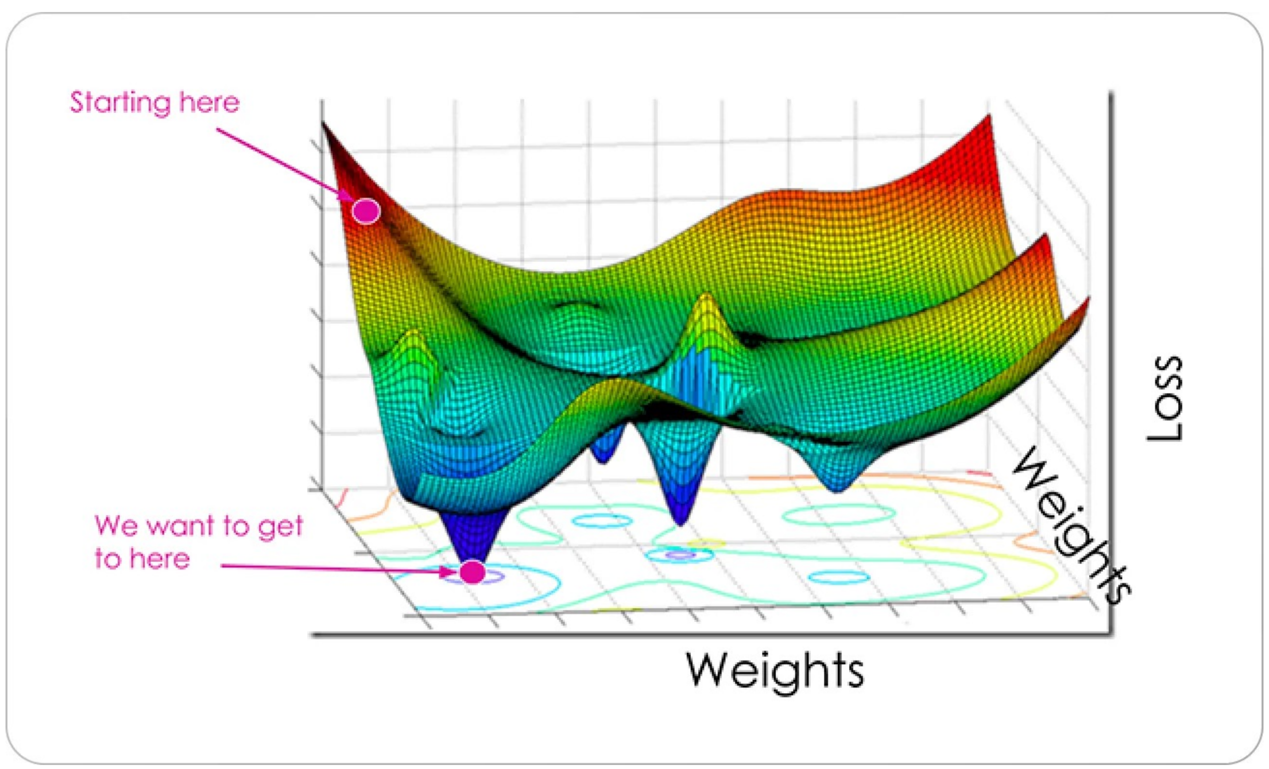
- For the hidden layer *j*, the output is calculated as

$$z = W \cdot X + b$$

  - ReLU: $\max(z, 0)$ Tanh: $\frac{e^z - e^{-z}}{e^z + e^{-z}}$
  - …
- For the output layer *k*, the output is calculated in a similar fashion to the hidden layers, common activation functions
  - Softmax for Multi-class Classification
  - Sigmoid for Binary Classification
  - Linear for Regression

Loss function

- The difference between the target and actual values arising at the output is referred to as the loss, and the associated function is the loss function
  - the goal of neural network learning process is to define these parameters in a way that the loss function is minimized
  - a feedback mechanism for adjusting the model's parameters
  - different numerical optimization algorithms can be used to determine weights and biases



Common Loss Functions

- Cross-entropy Loss for Classification
  - total cross-entropy loss over a dataset of *N* examples

$$\mathrm{L}(W, b) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c})$$

    - $\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c})$: the cross-entropy loss for a single example I
    - often combine with the softmax function in the output layer

- Mean Squared Error for Regression

$$\mathrm{L}(W, b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

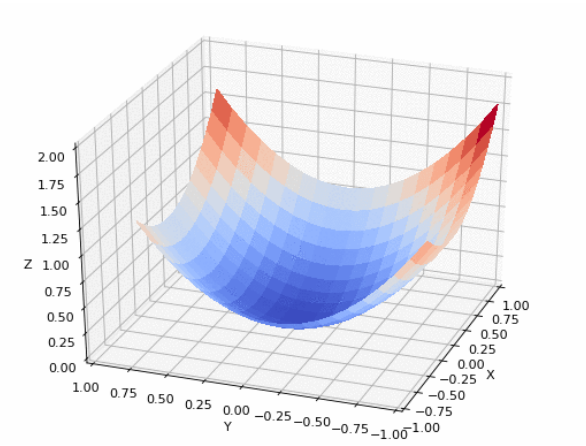
    - $(y_i - \hat{y}_i)^2$: square difference between the true label and the estimation on a single example i

Gradient Descent

- Optimisation in NNs primarily uses gradient descent
- Key idea: iteratively adjusting the model's parameters in the direction opposite to the gradient
  - the gradient points the most steeply direction
  - gradient is a vector of partial derivatives, each element represents how much the loss will be reduced by changing the weight

$$\nabla L(W_i) = \left[\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_p}\right]$$

  - why descent: finding a path from a randomly chosen starting point in the loss function that leads to the global minimum

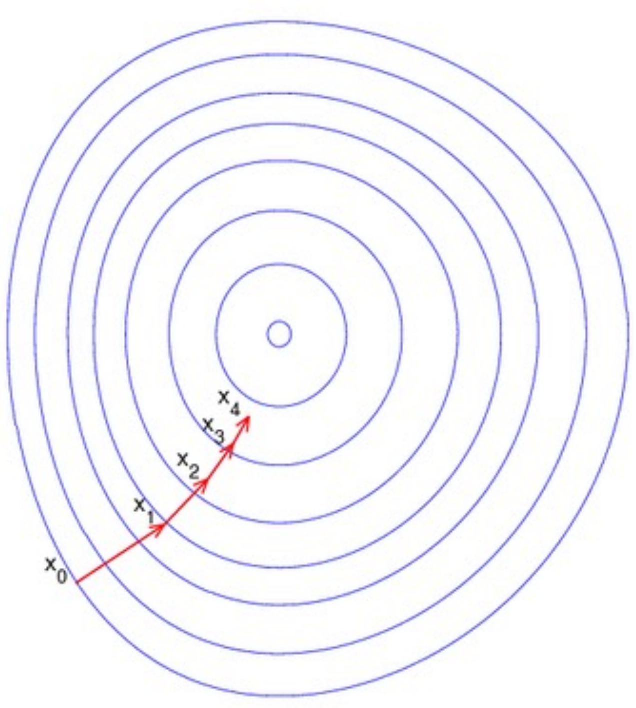


Gradient descent

- The update rule of gradient descent

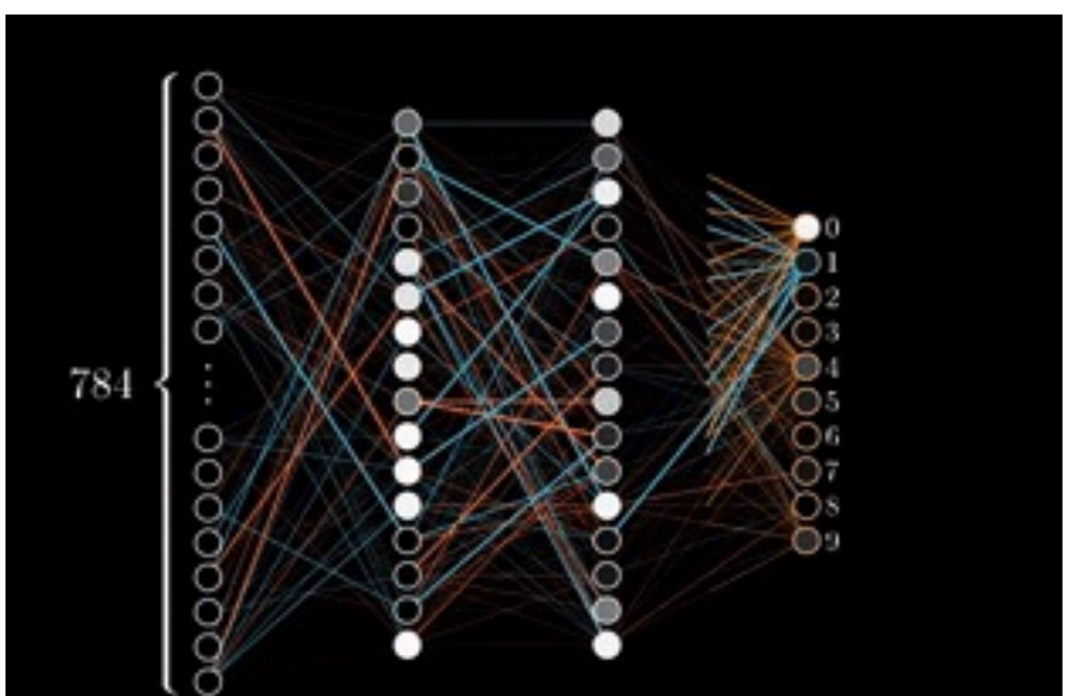
$$W_{i+1} = W_i - \eta \nabla L(W_i)$$

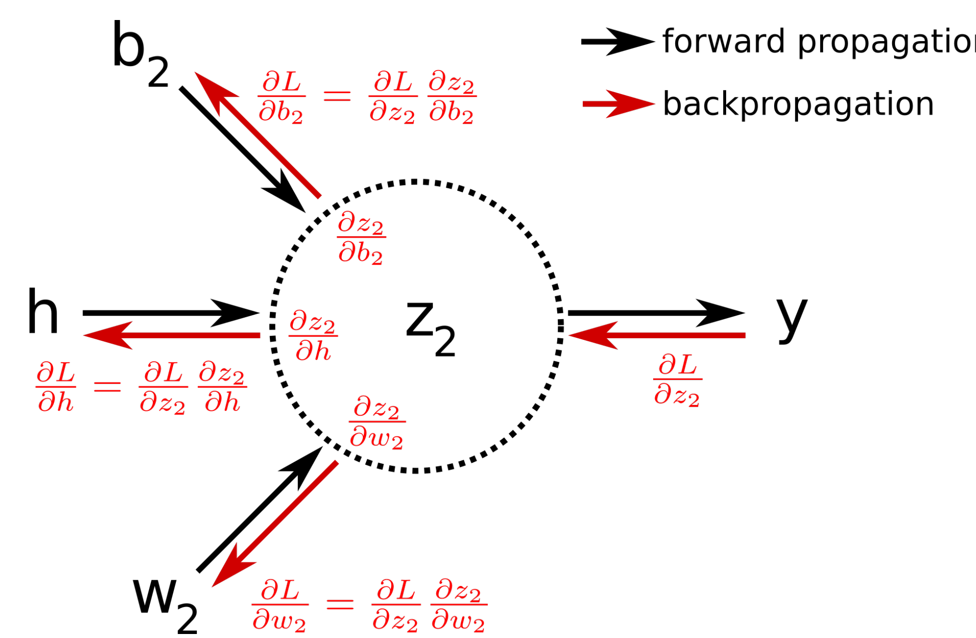
  - W represents the parameters of the model,
  - η is the learning rate, a positive value determining the update step size at each iteration. A higher learning rate makes the model learn faster;
  - $\nabla L(W_i)$ is the gradient of the loss function at the current parameters $\nabla L(W_i) = \left[\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \dots, \frac{\partial L}{\partial w_p}\right]$



Backpropagation

- Gradients are computed backwards
  - start at the output layer and compute the gradient of the loss function with respect to each output.
  - this typically involves finding how much a change in each output value affects the overall loss.
- Calculate the contribution of each weight to the loss function
- Backward propagating the gradient of the error



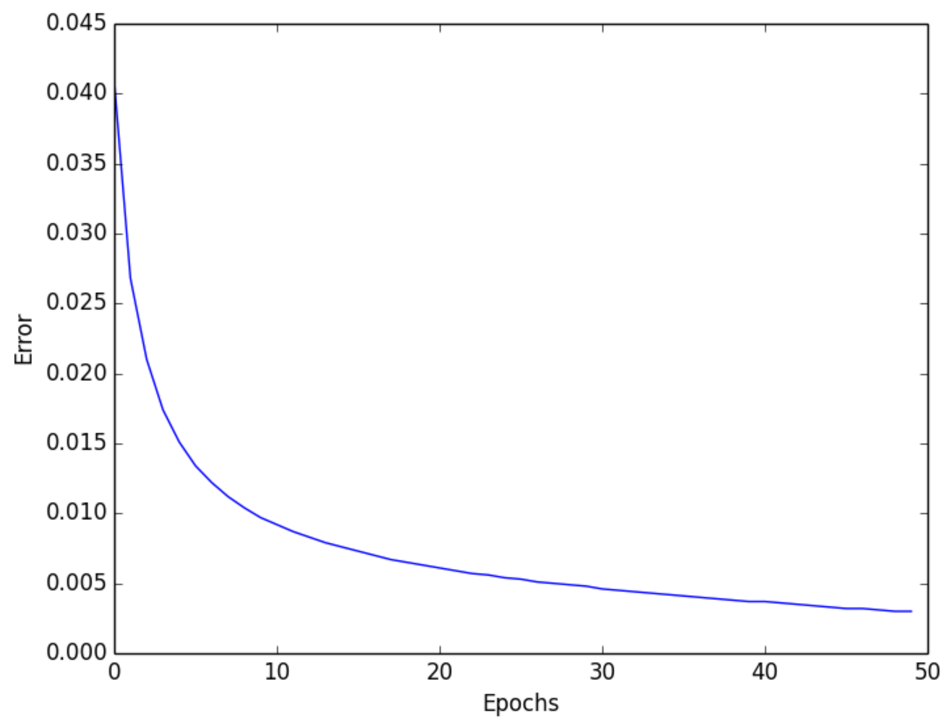


Backpropagation Algorithm

- Let η be the learning rate
- Set all weights to smaller random values
- Until total error is small enough, repeat
  - For each input example
    - Feed forward pass to get predicted outputs
    - Compute $\beta_z = d_z - o_z$ for each output node
    - Compute $\beta_j = \sum_k w_{j \to k} o_k (1 - o_k) \beta_k$
    - Compute the weight changes $\Delta w_{i \to j} = \eta o_i o_j (1 - o_j) \beta_j$
  - Add up weight changes for all input examples
  - Change weights

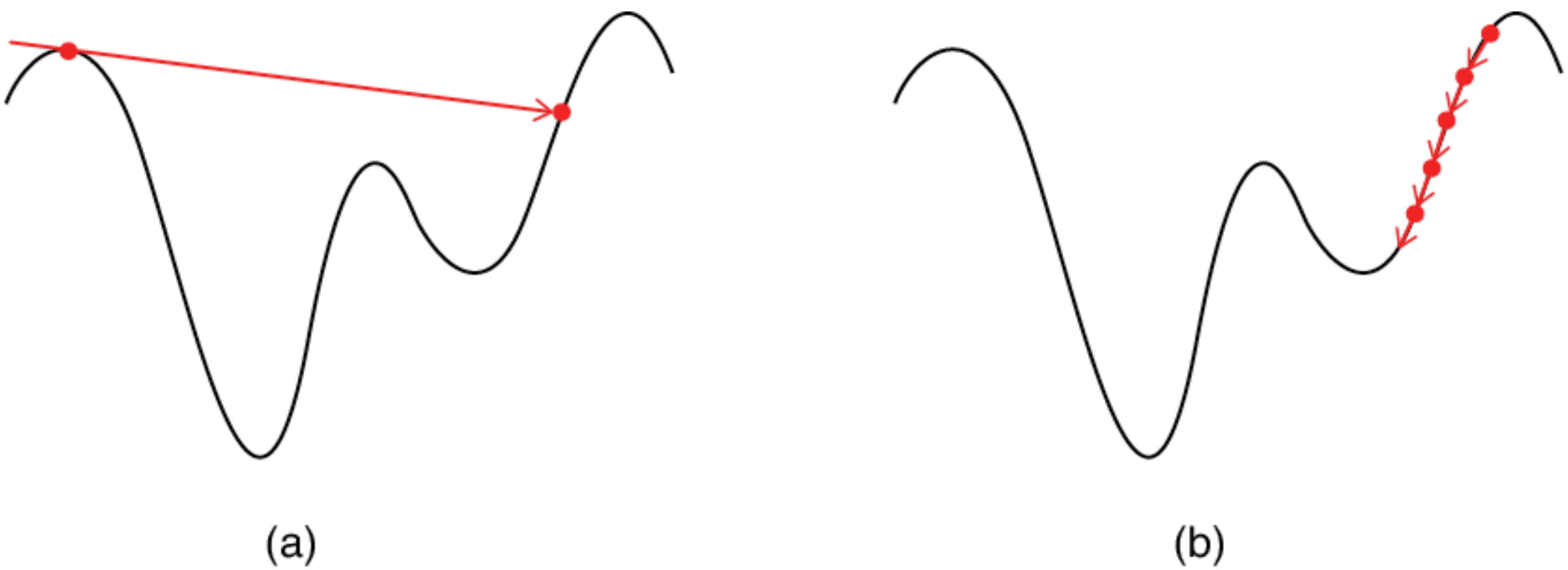
Notes on BP algorithm

- Convergence: The algorithm repeats until the error across the network does not improve significantly, or a predetermined number of epochs is reached.
- Epoch: one complete pass through the entire training dataset
- Too few epochs ->underfitting, too many epochs ->overfitting
- Training may require thousands of epochs
- A convergence curve will help to decide when to stop



Adjusting Learning Rate

- We can improve learning by changing the learning rate, however …



- When η is *too large (a), we can jump right over a deep valley*
- When η is *too small (b), we can slowly descend into a local minimum, and miss the deeper valley.*

Variants of Gradient Descents

- Stochastic/Mini Batch Gradient Descent: Gradient descent with minibatches, which uses subsets of the training data for each gradient descent step—the minibatch
  - Batch: the subset of training data used to update weights in one iteration.
  - Stochastic gradient descent with a batch size of one
  - using smaller minibatches often leads to models that perform better than those trained with larger minibatches
- Momentum: modify vanilla gradient descent to include a momentum term, a fraction of the previous step's update

$$v_t = \gamma v_{t-1} + \eta \nabla L(W_t)$$
$$w = w - \eta v_t$$

Adaptive learning rate methods

- AdaGrad: adapts the learning rate to the parameters, performing larger updates for infrequent parameters and smaller updates for frequent ones. Particularly useful for dealing with sparse data.

- RMSprop: a modification to Adagrad, works by maintaining a moving average of the squares of gradients and dividing the gradient by the square root of this average

- Adam(Adaptive Moment Estimation): combines the best properties of the AdaGrad and RMSprop, works by maintaining two moving averages for each parameter; one for the gradients (like RMSprop) and one for the square of the gradients (like AdaGrad). It then uses these estimates to adjust the learning rate for each parameter individually

Avoid Overfitting Through Regularisation

- Great flexibility of the network also means that it is prone to overfitting the training set
- The most popular regularization techniques for neural networks

  - Early Stopping**:** interrupt training when its performance on the validation set starts dropping

  - ℓ_1 and ℓ_2 Regularisation: modifying the cost function by adding $\lambda|W_t|$ or $\lambda||W_t||$

  - Data Augmentation: generating new training instances from existing ones, artificially boosting the size of the training set

Tools and Libraries

- TensorFlow (Google)
- Keras (integrated with TensorFlow , high-level API)
- PyTorch (Meta)
- Microsoft Cognitive Toolkit (CNTK): commercial-grade distributed deep learning
- Apache MXNet
- JAX (known for its ability to automatically differentiate)