Machine Learning and Photonics

Week 11

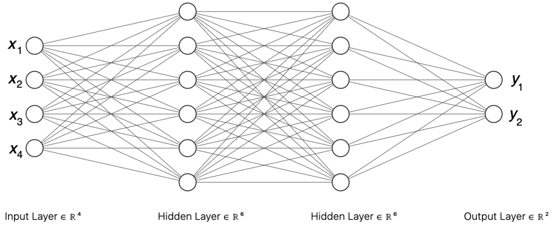
Ergun Simsek

University of Maryland Baltimore County

simsek@umbc.edu

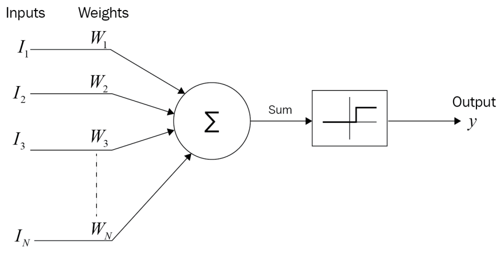
Fully Connected Neural Networks

April 4, 2023


x_1
x_2
x_3
x_4
y_1
y_2
Input Layer $\in \mathbb{R}^4$
Hidden Layer $\in \mathbb{R}^6$
Hidden Layer $\in \mathbb{R}^6$
Output Layer $\in \mathbb{R}^2$

Brief History of Neural Networks

The idea of NNs goes back to 1940s.



How do we learn? How the information is processed in brain?

- Threshold Logic: Continuous inputs generated discrete outputs
- Hebbian Learning: Cells that fire together, wire together
- 1958: McCulloch-Pitts neuron takes in inputs, takes a weighted sum, and returns "0" if the result is below the threshold and "1" otherwise.
- 1969: Minsky and Papert claimed that perceptrons can’t be effectively translated into multi-layered neural networks

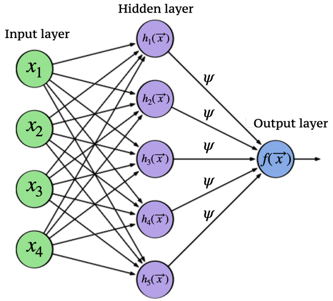
- 1982: Jon Hopfield ==> backpropagation
- Werbos, Rumelhart, Hinton, and Williams put these in a solid/clear framework
- Rebranding with a new name: deep learning.. Cheap memories. Generating extreme amount of data.. GPUs, TPUs, etc.

Deep Learning

- Fully-connected neural networks (FCNNs)
- Convolutional neural networks (CNNs)
- Recurrent neural networks (RNNs)
- Attention Mechanisms

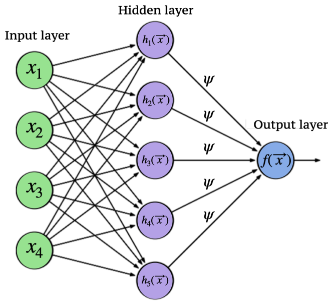
Basics of Neural Networks

- To illustrate how neural networks work, we start with a simple case.
- To set the stage, assume that we have d continuous features represented by $\vec{x} \in \mathbb{R}^d$, and a continuous target variable $y \in \mathbb{R}$.
- A neural network constructs a *nonlinear* model $f(\vec{x})$ to predict the target y.
- Neural networks are structured in a specific way and nonlinearity is introduced in the model in a peculiar way.



Basics of Neural Networks (Cont...)

- Every neural network possesses an input and an output layer.
- In the input layer (the green layer), the features of the model are fed into the model.
- In the output layer (the blue layer), the model offers its prediction for the target variable.



- The layers that are placed in between are called *hidden layers* (the purple layer).
- A neural network may have one or many hidden layers Each hidden layer consists of a number of units (the purple circles). Each unit of a hidden layer is called a **neuron**.

Basics of Neural Networks *(Cont...)*

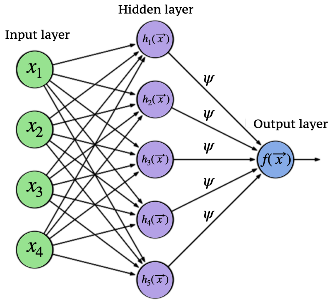
- The term feed-forward network means that all neurons feed their output forward to the next (hidden or output) layer, without any connections feeding to the same or the previous layer.
- The term fully-connected implies that each neuron in a hidden layer has a connection with (*i.e.* receives contribution from) **all** neurons in the previous layer.
- Any hidden layer - after receiving the output of its previous layer as a linear combination of the pervious neurons - applies an activation function (*i.e.* a nonlinear function) to the result.
- We'll talk about these activation functions in a few minutes.

How does a fully connected feed-forward neural network predict the target?

- For simplicity assume that we have only one hidden layer with N neurons.
- The predicted value of the target variable y is given by the following formula

$$\hat{y}^{(i)} = f(\vec{x}^{(i)}) = \alpha_0 + \sum_{j=1}^{N} \alpha_j h_j(\vec{x}^{(i)})$$

$$= \alpha_0 + \sum_{j=1}^{N} \alpha_j \, \psi\Big(\omega_{j0} + \sum_{k=1}^{d} \omega_{jk} x_k^{(i)}\Big) \; . \tag{1}$$



How does a fully connected feed-forward neural network predict the target? *(Cont...)*

- Each neuron h_j is constructed as a linear combination of *all* constituents of the previous layer
- Then the activation function ψ is applied to the result of each neuron
- The linear combination of contributions coming from *all* neurons determines the predicted value of the target $\hat{y}^{(i)}$ for the i-th sample.

$$\begin{aligned}\hat{y}^{(i)} = f(\vec{x}^{(i)}) &= \alpha_0 + \sum_{j=1}^{N} \alpha_j h_j(\vec{x}^{(i)}) \\ &= \alpha_0 + \sum_{j=1}^{N} \alpha_j\, \psi\big(\omega_{j0} + \sum_{k=1}^{d} \omega_{jk} x_k^{(i)}\big)\ .\end{aligned} \tag{1}$$

How does a fully connected feed-forward neural network predict the target? *(Cont...)*

Question: What are the parameters of the above model? How many parameters does the above fully connected neural network possess?
The parameters of the model are the coefficients $\alpha = (\alpha_0, \alpha_1, \cdots, \alpha_N)$ and ω_{jk} that can be cast in the following $N \times (d+1)$ matrix

$$\mathbb{W} = \begin{pmatrix} \omega_{10} & \omega_{11} & \omega_{12} & \cdots & \omega_{1d} \\ \omega_{20} & \omega_{21} & \omega_{22} & \cdots & \omega_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_{N0} & \omega_{N1} & \omega_{N2} & \cdots & \omega_{Nd} \end{pmatrix}. \tag{2}$$

How does a fully connected feed-forward neural network predict the target? *(Cont...)*

The columns of matrix $\mathbb{W}$ refer to different neurons.
The rows of $\mathbb{W}$ refer to the d features and the bias terms.

IOW, ω_{jk} with $k \geq 1$ is the weight the j-th neuron receives from the k-th feature x_k.

Hence an FCNN with a single hidden layer possesses

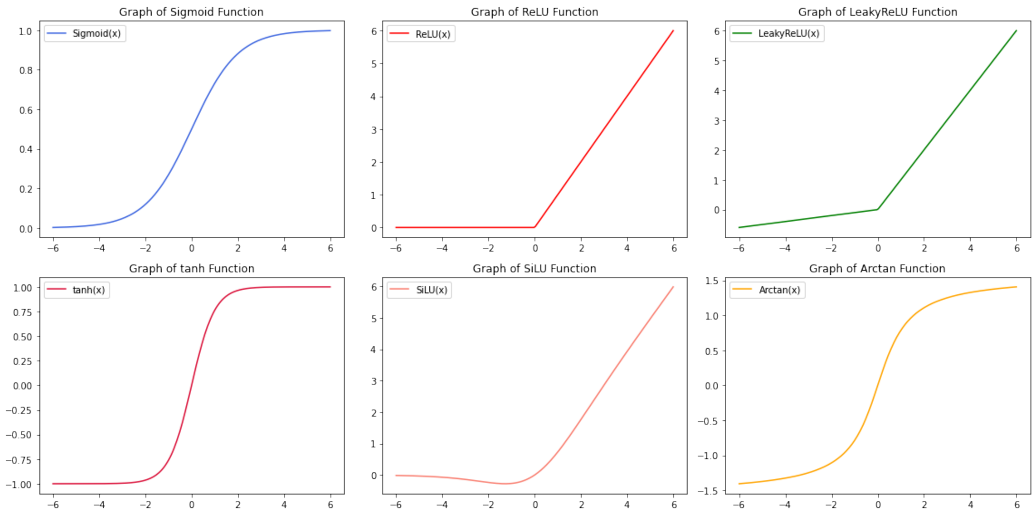
$$\text{Total number of weights } = N(d+1) + N + 1 = N(d+2) + 1 \tag{3}$$

parameters.

(Most Common) Activation Functions

Sigmoid	$\sigma(x) = \dfrac{1}{1+e^{-x}}$
Rectified Linear Unit	$ReLU(x) = \begin{cases} 0\,, & x < 0 \\ x\,, & x \geq 0 \end{cases}$
Leaky Rectified Linear Unit	$ReLU_{\alpha}(x) = \begin{cases} \alpha x\,, & x < 0 \\ x\,, & x \geq 0 \end{cases}$
Hyperbolic Tangent	$\tanh(x) = \dfrac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
Sigmoid Linear Unit	$SiLU(x) = x\,\sigma(x) = \dfrac{x}{1+e^{-x}}$
Arc-tangent	$\arctan(x) = \tan^{-1}(x)$

(Most Common) Activation Functions



Why Do Activation Functions Do?

To illustrate how this phenomenon occurs, consider the following very elementary example. Suppose we have only two features $\vec{x} = (x_1, x_2)$, and we design a neural network with only one hidden layer with two neurons. We use the activation $\psi(x) = \text{arctan}(x)$. Consider the following values for the weights of this simple model

$$\alpha = (\alpha_0, \alpha_1, \alpha_2) = (0, \frac{1}{2}, \frac{1}{2}) , \qquad \mathbb{W} = \begin{pmatrix} \omega_{10} & \omega_{11} & \omega_{12} \\ \omega_{20} & \omega_{21} & \omega_{22} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & -1 \end{pmatrix} . \tag{4}$$

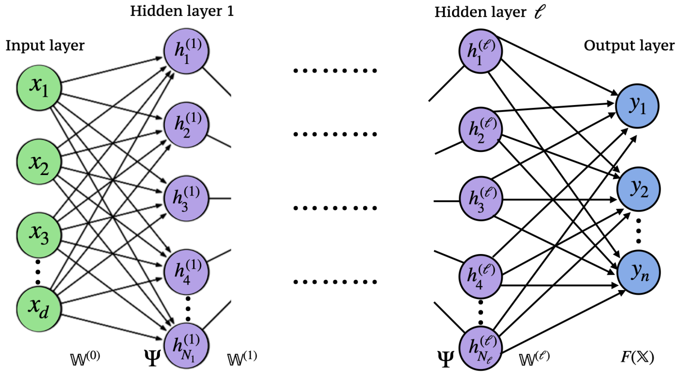
Why Do Activation Functions Do? *(Cont...)*

Then, the corresponding neural network predicts the target variable through the following formula

$$
\begin{aligned}
\hat{y} = f(\vec{x}) &= \alpha_0 + \sum_{j=1}^{2} \alpha_j \, \psi\Big(\omega_{j0} + \sum_{k=1}^{d} \omega_{jk} x_j^{(i)}\Big) \\
&= \frac{1}{2}\psi(x_1 + x_2) + \frac{1}{2}\psi(x_1 - x_2) = \frac{1}{2}\Big(\arctan(x_1 + x_2) + \arctan(x_1 - x_2)\Big) \\
&= x_1 - x_1 x_2^2 - \frac{x_1^3}{3} + 2\, x_1^3 x_2^2 + x_1 x_2^4 + \frac{x_1^5}{5} + \cdots
\end{aligned}
\tag{5}
$$

Multiple Hidden Layers

Suppose we have *d* continuous features, $\vec{x} \in \mathbb{R}^d$ & one cont. target variable $y \in \mathbb{R}^n$.
Assume we have an FCNN with ℓ hidden layers.
Each hidden layer *i* consists of N_i neurons.



Multiple Hidden Layers *(Cont...)*

Then, the prediction of the this neural network for the target variable y is given by

$$\hat{\mathbb{Y}} = F(\mathbb{X}) = \underbrace{\mathbb{W}^{(\ell)}\Psi\Big(\mathbb{W}^{(\ell-1)}\Psi\cdots\Psi\big(\mathbb{W}^{(1)}\Psi}_{\ell \text{ times}}(\mathbb{W}^{(0)}\mathbb{X}^{\mathsf{T}})\big)\cdots\Big) = \Big(\prod_{j=1}^{\ell}\mathbb{W}^{(j)}\circ\Psi\Big)(\mathbb{W}^{(0)}\mathbb{X}^{\mathsf{T}}), \tag{6}$$

where the weight matrix $\mathbb{W}^{(i)}$, for $0 \leq i \leq \ell$, is an $N_{i+1} \times (N_i + 1)$ matrix that encodes the weights of the i-th hidden layer

Multiple Hidden Layers *(Cont...)*

$$\mathbb{W}^{(i)} = \begin{pmatrix} \omega_{10}^{(i)} & \omega_{11}^{(i)} & \omega_{12}^{(i)} & \cdots & \omega_{1N_i}^{(i)} \\ \omega_{20}^{(i)} & \omega_{21}^{(i)} & \omega_{22}^{(i)} & \cdots & \omega_{2N_i}^{(i)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_{N_{i+1}0}^{(i)} & \omega_{N_{i+1}1}^{(i)} & \omega_{N_{i+1}2}^{(i)} & \cdots & \omega_{N_{i+1}N_i}^{(i)} \end{pmatrix}, \tag{7}$$

where $N_0 = d$ and $N_{\ell+1} = n$.

Note that in Eq. (6), Ψ acts on a given matrix component-wise (*i.e.* the action of Ψ on a matrix is the same as acting the activation function ψ on all elements of the matrix).

Multiple Hidden Layers (Cont...)

- In principle, one can even apply different activation functions for different hidden layers.
- This can be easily accommodated in Eq. (6) by considering a subscript for each activation function (*i.e.* replace Ψ in the right hand side of (6) by $\Psi^{(j)}$).
- A short analysis reveals that the total number of weights of the model is given by

$$\text{Total number of weights} = \sum_{i=0}^{\ell} N_{i+1}(N_i + 1) \,, \tag{8}$$

NNs Grow Fast

In order to obtain a sense about the typical number of parameters involved a neural network, let us set $d = 10$, $n = 1$, and assume we consider 3 hidden layers each with 256 neurons (a very common size for a hidden layer in deep learning).

Then, according to (8), the total number of parameters of such neural network is given by

$$\begin{aligned} \text{Total number of weights} &= \sum_{i=0}^{3} N_{i+1}(N_i + 1) \\ &= 256 \times (10 + 1) + 256 \times (256 + 1) \\ &\quad + 1 \times (256 + 1) \\ &= 134657\ . \end{aligned} \tag{9}$$

Even the simplest neural networks involve a large number of parameters.
This implies that neural networks possess a *large number of degrees of freedom*, and hence, they can potentially *suffer from an overfitting problem*.

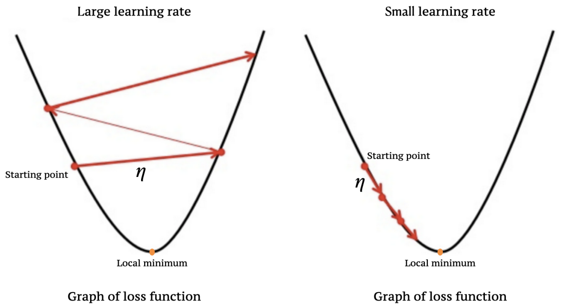
Training Neural Networks

- Due to the large of number of parameters (weights) a neural network involves, we use Gradient Descents for training neural networks.
- In the context of ML, the small step in gradient descent by which one moves in the opposite direction of the gradient is called the **learning rate**.
- In order to find the optimal values of the weights of a neural network, an appropriate cost (loss) function is optimized.
- Given a loss function $\mathcal{L}$, the optimal values for weights Ω are found by minimizing the loss function.
- Throughout the optimization process, the parameters at each step k are updated as follows

$$\Omega \longleftarrow \Omega - \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \vec{\nabla}_{\Omega} \mathcal{L}\left(y^{(i)}, f_{\Omega}(\vec{x}^{(i)})\right) . \tag{11}$$

Training Neural Networks

- If the target variable of the neural network is continuous, a typical choice (there are several choices) for the loss function is the usual *MSE* (mean squared error) function.
- If the target variable of the neural network is a categorical variable, then an appropriate choice for the loss function would be the *cross-entropy* function.



- In general the loss function is a non-convex function of the weights of the NN, i.e. the solution to the minimization problem of the loss function may not be unique, and there may exist multiple local minima for the loss function.
- Due to the high number of weights a typical NN involves, NNs are in danger of suffering from an overfitting problem.
    - To reduce the chance of overfitting, the train data is fit to the model in an iterative process with small steps. Whenever signs of high variance are detected, the training is stopped. This procedure is usually referred to as the **slow learning**.
    - Another method to deal with overfitting in neural networks is to take the advantage of *regularization methods* we introduced earlier (*e.g.* L_1 and L_2 regularizations).

Question: The execute the gradient descent algorithm, we *need a starting point* (*i.e.* point P). How is the starting point in the parameter space picked?

Answer: The gradient descent algorithm that is built in deep learning models automatically picks a *random point* in the parameter space. This means that the coordinates of the starting point (*i.e.* initial values of the parameters of the model) are typically picked from a Gaussian distribution with mean 0.

Question: Given the complexity of neural networks, how are the gradients practically calculated?

Answer: Chain rule!

Back-Propagation

- To show how the calculation of the gradients of the loss function is implemented, let's reconsider our one layer NN.
- The prediction of the model is given by function f given in Eq. (1).
- This neural network has two sets of weights, namely α_j and ω_{jk}.
- We indicate these $N(d+2)+1$ weights collectively by Ω.
- Suppose, we choose the loss function $\mathcal{L}$ to be the *MSE* function. Then

$$\begin{aligned}\mathcal{L}(\Omega) = \sum_{i=1}^{N_{train}} \mathcal{L}^{(i)}(\Omega) &= \frac{1}{2}\sum_{i=1}^{N_{train}} (y^{(i)} - \hat{y}^{(i)})^2 = \frac{1}{2}\sum_{i=1}^{N_{train}} \left(y^{(i)} - f_\Omega(\vec{x}^{(i)})\right)^2 \\ &= \frac{1}{2}\sum_{i=1}^{N_{train}} \left[y^{(i)} - \left(\alpha_0 + \sum_{j=1}^{N} \alpha_j\, \psi(z_j^{(i)})\right)\right]^2 ,\end{aligned} \tag{12}$$

where $z_j^{(i)} = \omega_{j0} + \sum_{k=1}^{d} \omega_{jk} x_k^{(i)}$.

Back-Propagation *(Cont...)*

The gradient of the loss function consists of two sets of derivatives:

$$\vec{\nabla}_{\Omega}\mathcal{L}(\Omega) = \langle \partial_{\alpha_i}\mathcal{L}, \partial_{\omega_{jk}}\mathcal{L} \rangle \ , \qquad \text{for } i, j = 0, 1, \cdots N, \quad \text{and } k = 0, 1, \cdots d \ . \tag{13}$$

In order to calculate the partial derivatives in (13), we need to use the *chain rule* of the multivariable calculus.

Back-Propagation (Cont...)

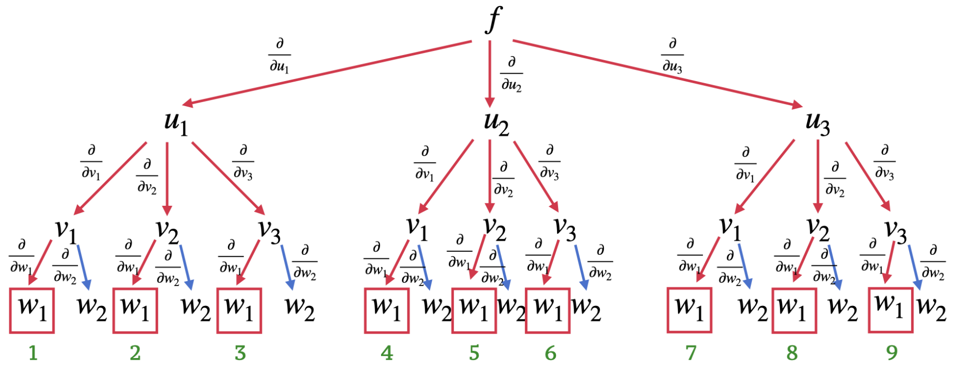
Using the chain rule, the partial derivatives of the loss function can be easily computed as follows:

$$
\begin{aligned}
\partial_{\alpha_0}\mathcal{L}(\Omega) &= \frac{\partial \mathcal{L}}{\partial \alpha_0} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial \alpha_0} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial f_\Omega(\vec{x}^{(i)})} \frac{\partial f_\Omega(\vec{x}^{(i)})}{\partial \alpha_0} = -\sum_{i=1}^{N_{train}} f_\Omega(\vec{x}^{(i)}) \,, \\
\partial_{\alpha_j}\mathcal{L}(\Omega) &= \frac{\partial \mathcal{L}}{\partial \alpha_j} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial \alpha_j} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial f_\Omega(\vec{x}^{(i)})} \frac{\partial f_\Omega(\vec{x}^{(i)})}{\partial \alpha_j} = -\sum_{i=1}^{N_{train}} f_\Omega(\vec{x}^{(i)})\, \psi(z_j^{(i)}) \,, \\
\partial_{\omega_{j0}}\mathcal{L}(\Omega) &= \frac{\partial \mathcal{L}}{\partial \omega_{j0}} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial \omega_{j0}} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial f_\Omega(\vec{x}^{(i)})} \frac{\partial f_\Omega(\vec{x}^{(i)})}{\partial \psi(z_j^{(i)})} \frac{\partial \psi(z_j^{(i)})}{\partial z_j^{(i)}} \frac{\partial z_j^{(i)}}{\partial \omega_{j0}} = -\sum_{i=1}^{N_{train}} f_\Omega(\vec{x}^{(i)})\, \alpha_j\, \psi'(z_j^{(i)}) \,, \\
\partial_{\omega_{jk}}\mathcal{L}(\Omega) &= \frac{\partial \mathcal{L}}{\partial \omega_{jk}} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial \omega_{jk}} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial f_\Omega(\vec{x}^{(i)})} \frac{\partial f_\Omega(\vec{x}^{(i)})}{\partial \psi(z_j^{(i)})} \frac{\partial \psi(z_j^{(i)})}{\partial z_j^{(i)}} \frac{\partial z_j^{(i)}}{\partial \omega_{jk}} = -\sum_{i=1}^{N_{train}} f_\Omega(\vec{x}^{(i)})\, \alpha_j\, \psi'(z_j^{(i)})\, x_k^{(i)} \,.
\end{aligned}
\tag{14}
$$

Back-Propagation *(Cont...)*

- Equation (14) shows how the gradient of the loss function is computed as the sum of individual contributions from the samples.
- To calculate the contribution of each sample, the chain rule of the multivariable calculus has to be applied.
- The process of calculating all necessary partial derivatives of the loss function using the chain rule is known as the **backpropagation** in the context of deep learning.
- Backpropagation is a precise bookkeeping tool to calculate all necessary partial derivatives of the loss function with respect to various weights of the neural network.

Back-Propagation (Cont...)



f
$\frac{\partial}{\partial u_1}$
$\frac{\partial}{\partial u_2}$
$\frac{\partial}{\partial u_3}$
u_1
u_2
u_3
$\frac{\partial}{\partial v_1}$
$\frac{\partial}{\partial v_2}$
$\frac{\partial}{\partial v_3}$
v_1
v_2
v_3
$\frac{\partial}{\partial w_1}$
$\frac{\partial}{\partial w_2}$
w_1
w_2
1
2
3
4
5
6
7
8
9

$$\frac{\partial f}{\partial w_1} = \overbrace{\frac{\partial v_1}{\partial w_1}\frac{\partial u_1}{\partial v_1}\frac{\partial f}{\partial u_1}}^{1} + \overbrace{\frac{\partial v_2}{\partial w_1}\frac{\partial u_1}{\partial v_2}\frac{\partial f}{\partial u_1}}^{2} + \overbrace{\frac{\partial v_3}{\partial w_1}\frac{\partial u_1}{\partial v_3}\frac{\partial f}{\partial u_1}}^{3} + \overbrace{\frac{\partial v_1}{\partial w_1}\frac{\partial u_2}{\partial v_1}\frac{\partial f}{\partial u_2}}^{4} + \overbrace{\frac{\partial v_2}{\partial w_1}\frac{\partial u_2}{\partial v_2}\frac{\partial f}{\partial u_2}}^{5}$$

$$+ \underbrace{\frac{\partial v_3}{\partial w_1}\frac{\partial u_2}{\partial v_3}\frac{\partial f}{\partial u_2}}_{6} + \underbrace{\frac{\partial v_1}{\partial w_1}\frac{\partial u_3}{\partial v_1}\frac{\partial f}{\partial u_3}}_{7} + \underbrace{\frac{\partial v_2}{\partial w_1}\frac{\partial u_3}{\partial v_2}\frac{\partial f}{\partial u_3}}_{8} + \underbrace{\frac{\partial v_3}{\partial w_1}\frac{\partial u_3}{\partial v_3}\frac{\partial f}{\partial u_3}}_{9}$$

NNs and Python

- Pytorch
  - Developed by Facebook’s AI Research team
  - Known for its ease of use, flexibility, and dynamic computational graph construction
  - Uses a technique called autograd to automatically compute gradients
  - Has a large collection of pre-trained models
- Keras-Tensorflow
  - Developed by Google
  - Known for its ease of use, flexibility, and static computational graph construction
  - Has a wide collection of pre-trained models
  - Slow(ish) compared to Pythorch

A Gentle Introduction to Tensors

- In PyTorch, tensors are the basic building blocks of deep learning models.
- Tensors are multi-dimensional arrays that can be used to represent various types of data, including images, text, and numerical data.
- PyTorch provides a user-friendly interface for defining and manipulating tensors, which makes it easy for researchers and data scientists to build deep learning models.

A Gentle Introduction to Tensors

To create a tensor in PyTorch, you can use the torch.tensor() function. This function takes a list or a NumPy array as input and returns a PyTorch tensor.

```
import torch

# create a tensor from a list
my_list = [1, 2, 3]
my_tensor = torch.tensor(my_list)

# create a tensor from a NumPy array
import numpy as np
my_array = np.array([4, 5, 6])
my_tensor = torch.tensor(my_array)
```

A Gentle Introduction to Tensors *(Cont...)*

PyTorch tensors can be manipulated using a variety of operations, including arithmetic operations, matrix operations, and comparison operations. PyTorch provides many built-in functions to perform these operations.

```
# create two tensors
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])

# perform arithmetic operations
# c = a + b,  d = a - b, e = a * b, f = a / b

# perform matrix operations like dot prod or matrix multip.
g = torch.dot(a, b)
h = torch.mm(a.reshape(1, 3), b.reshape(3, 1))

# perform comparison operations
i = (a > b)
j = (a == b)
```

A Gentle Introduction to Tensors *(Cont...)*

When a tensor is created with the requires_grad parameter set to True, PyTorch tracks all the operations performed on the tensor and creates a computational graph. This computational graph is then used to compute the gradients of the loss function with respect to the tensor.

```
# create a tensor with requires_grad set to True
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)

# define a function
def y_fn(x):
    y = x**2 + 2*x + 1
    return y

# compute the output and the gradient
y = y_fn(x)
y.backward()

# print the gradient
print(x.grad)
```

A Great Reference To Learn NNs with Pytorch

- Edward Raff, *Inside Deep Learning, Math, Algorithms, Models*, Manning Publications (2021), `https://www.manning.com/books/inside-deep-learning`

Let's look at a few examples!