Neural Networks

— The core tool of “deep learning” —

Jonathan Arsenault, Charles C. Cossette, Vassili Korotkin

McGill University, Department of Mechanical Engineering

November 7, 2022

Motivation

"The transition from classical computer vision techniques to the deep learning approach was a huge bump up in performance. The results we've seen there are way beyond anything we got with the classical techniques . . .

—Adam Bry, CEO of Skydio

A leader in autonomous drones.

`https://youtu.be/ncZmnfIRIWE`

Motivation

"The transition from classical computer vision techniques to the deep learning approach was a huge bump up in performance. The results we've seen there are way beyond anything we got with the classical techniques ...

... where we see the most success is in applying a deep understanding of the first principles and physics of the problem, in order to craft the learning into exploiting the structure of the problem."

—Adam Bry, CEO of Skydio

A leader in autonomous drones.

`https://youtu.be/ncZmnfIRIWE`

Neural Networks in Robotics

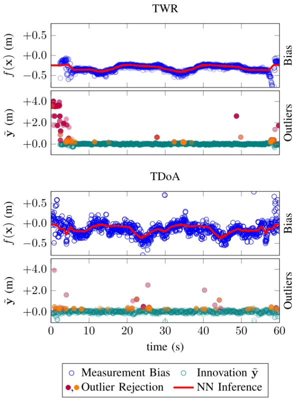


Figure 1: Learning UWB Bias [1]

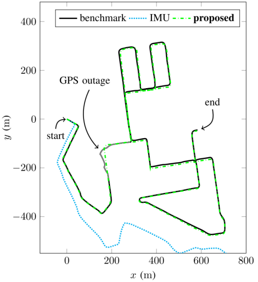


Figure 2: AI-IMU Dead reckoning [3]

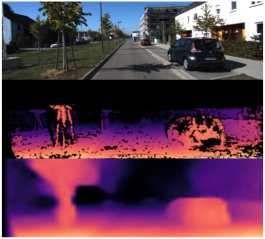


Figure 3: (top) Raw image. (middle) LIDAR depth. (bottom) Learned depth.

MPC Acceleration [2]:

	Computation Time
MPC	6.75 ms
NN MPC	0.00584 ms

Recall Linear Regression

- Recall the problem of fitting a line to some data. For sample i, we measure both the **input** $\mathbf{x}^{(i)}$ and corresponding **output** $y^{(i)}$. This creates our dataset

$$\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})\}, \tag{1}$$

where $\mathbf{x} = [x_1 \ldots x_D]^\mathsf{T}$.

Recall Linear Regression

- Recall the problem of fitting a line to some data. For sample i, we measure both the **input** $\mathbf{x}^{(i)}$ and corresponding **output** $y^{(i)}$. This creates our dataset

$$\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})\}, \tag{1}$$

  where $\mathbf{x} = [x_1 \ldots x_D]^\mathsf{T}$.

- We would like to fit the following simple model to this data

$$y = w_1 x_1 + w_2 x_2 + \ldots + w_D x_D + b = \underbrace{\mathbf{w}^\mathsf{T}\mathbf{x} + b}_{f(\mathbf{x}, \boldsymbol{\theta})}. \tag{2}$$

- The problem is to find the **parameters** $\boldsymbol{\theta} = (\mathbf{w}, b)$ that "best fit" the data.

Recall Linear Regression

- Recall the problem of fitting a line to some data. For sample i, we measure both the **input** $\mathbf{x}^{(i)}$ and corresponding **output** $y^{(i)}$. This creates our dataset

$$\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(N)}, y^{(N)})\}, \tag{1}$$

  where $\mathbf{x} = [x_1 \dots x_D]^\mathsf{T}$.

- We would like to fit the following simple model to this data

$$y = w_1 x_1 + w_2 x_2 + \dots + w_D x_D + b = \underbrace{\mathbf{w}^\mathsf{T}\mathbf{x} + b}_{f(\mathbf{x},\boldsymbol{\theta})}. \tag{2}$$

- The problem is to find the **parameters** $\boldsymbol{\theta} = (\mathbf{w}, b)$ that "best fit" the data.
- This can be done by minimizing a *loss (cost) function*

$$L(\boldsymbol{\theta}, \mathcal{D}) = \frac{1}{2N} \sum_{i=1}^{N} \left\| y^{(i)} - f(\mathbf{x}^{(i)}, \boldsymbol{\theta}) \right\|^2, \qquad N = |\mathcal{D}|. \tag{3}$$

- For the model in (2), problem (3) is solved analytically with least squares.

Generalizing Regression

- We can take this idea of having a model, with a bunch of parameters to optimize, but make the model nonlinear.
- As such, neural networks can accomplish exactly the same task. They are just another (nonlinear) function

$$y = f_{\mathrm{NN}}(\mathbf{x}, \boldsymbol{\theta}) \tag{4}$$

  which has parameters that we must optimize to accomplish a specific task.
- Neural networks can also predict multiple outputs at once

$$\mathbf{y} = \mathbf{f}_{\mathrm{NN}}(\mathbf{x}, \boldsymbol{\theta}). \tag{5}$$

Regression vs Classification

Predict output variable y from input variable $\mathbf{x}$.

Regression

Output $y \in \mathbb{R}$ is continuous. Example: Linear regression.

Classification

Output y belongs to one of K classes, $y \in \{y_1, \ldots, y_K\}$. Example: Clustering.

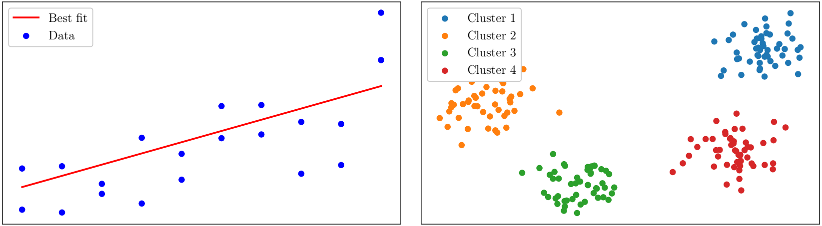
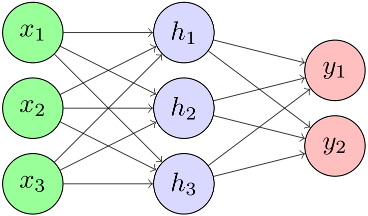
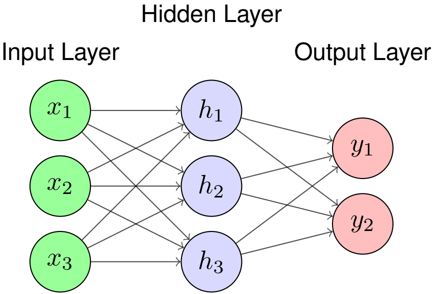


Figure 4: Left: Linear regression illustration, Right: Clustering

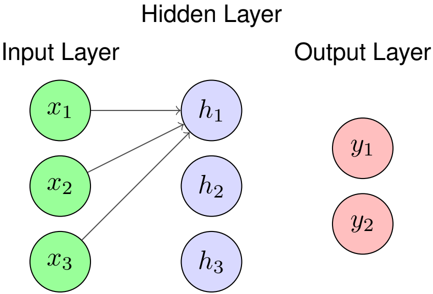
A Simple Neural Network



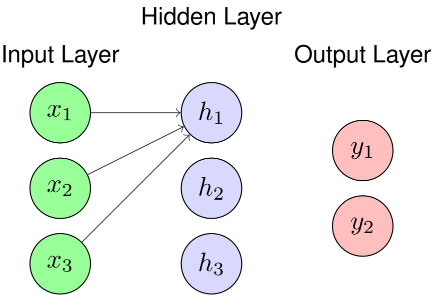
A Simple Neural Network



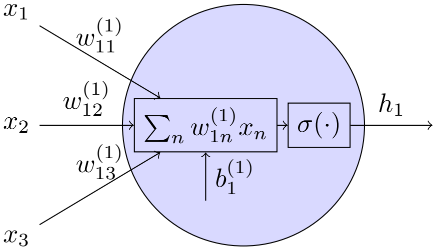
A Simple Neural Network



A Simple Neural Network



A Single Neuron Example:

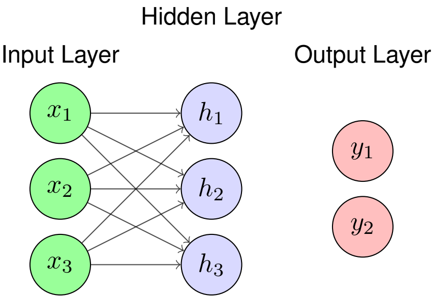


- Output of neuron h_1 is

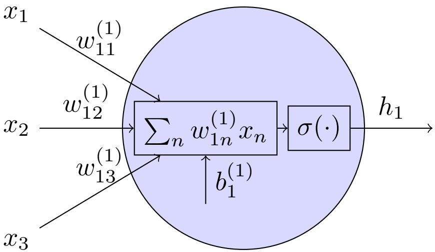
$$h_1 = \sigma\Big(\sum_n w_{1n}^{(1)} x_n + b_1^{(1)}\Big) \tag{6}$$

$$\triangleq \sigma\big(\mathbf{w}_1^{(1)^\mathsf{T}} \mathbf{x} + b_1^{(1)}\big). \tag{7}$$

A Simple Neural Network



A Single Neuron Example:



- Output of neuron h_1 is

$$h_1 = \sigma(\sum_n w_{1n}^{(1)} x_n + b_1^{(1)}) \quad (6)$$

$$\triangleq \sigma(\mathbf{w}_1^{(1)\mathsf{T}} \mathbf{x} + b_1^{(1)}). \quad (7)$$

- For an entire layer,

$$\mathbf{h} = \begin{bmatrix} h_1 \\ \vdots \\ h_M \end{bmatrix} = \begin{bmatrix} \sigma(\mathbf{w}_1^{(1)\mathsf{T}} \mathbf{x} + b_1^{(1)}) \\ \vdots \\ \sigma(\mathbf{w}_M^{(1)\mathsf{T}} \mathbf{x} + b_M^{(1)}) \end{bmatrix}$$

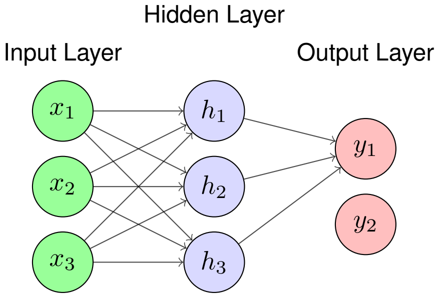
$$\triangleq \boldsymbol{\sigma}(\mathbf{W}^{(1)} \mathbf{x} + \mathbf{b}^{(1)}) \quad (8)$$

where

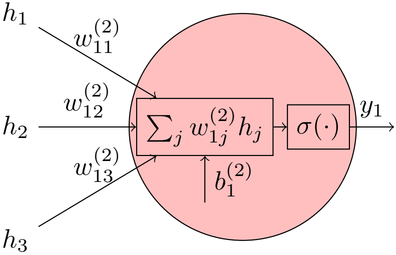
$$\mathbf{W}^{(1)} = \begin{bmatrix} \mathbf{w}_1^{(1)\mathsf{T}} \\ \vdots \\ \mathbf{w}_M^{(1)\mathsf{T}} \end{bmatrix}, \mathbf{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ \vdots \\ b_M^{(1)} \end{bmatrix}$$

and $\boldsymbol{\sigma}(\cdot)$ is just the element-wise application of $\sigma(\cdot)$.

A Simple Neural Network



A Single Neuron Example:

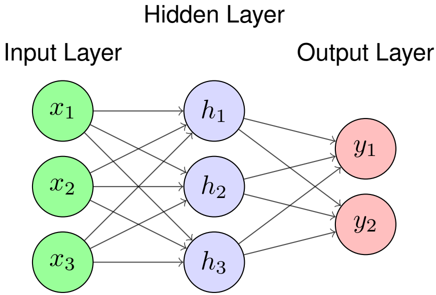


- Output of neuron y_1 is

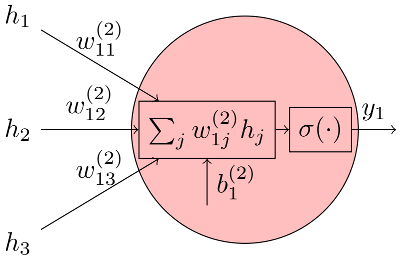
$$\hat{y}_1 = \sigma\left(\sum_j w_{1j}^{(2)} h_j + b_1^{(2)}\right) \tag{9}$$

$$\triangleq \sigma(\mathbf{w}_1^{(2)^\top} \mathbf{h} + b_1^{(2)}). \tag{10}$$

A Simple Neural Network



A Single Neuron Example:



- Output of neuron y_1 is

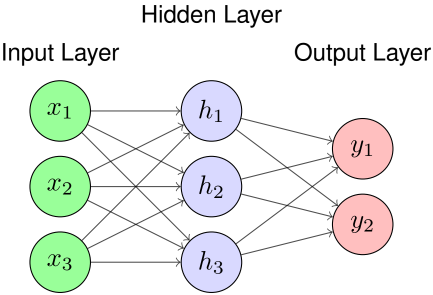
$$\hat{y}_1 = \sigma\left(\sum_j w_{1j}^{(2)} h_j + b_1^{(2)}\right) \tag{9}$$

$$\triangleq \sigma(\mathbf{w}_1^{(2)^\top} \mathbf{h} + b_1^{(2)}). \tag{10}$$

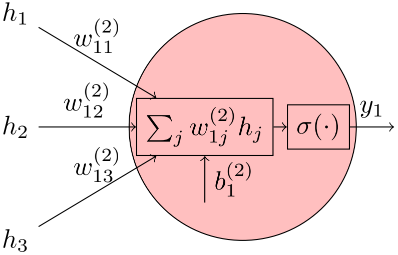
- Output layer is computed with

$$\hat{\mathbf{y}} = \boldsymbol{\sigma}(\mathbf{W}^{(2)} \mathbf{h} + \mathbf{b}^{(2)}) \tag{11}$$

A Simple Neural Network



A Single Neuron Example:



- Output of neuron y_1 is

$$\hat{y}_1 = \sigma\left(\sum_j w_{1j}^{(2)} h_j + b_1^{(2)}\right) \tag{9}$$

$$\triangleq \sigma(\mathbf{w}_1^{(2)\mathsf{T}}\mathbf{h} + b_1^{(2)}). \tag{10}$$

- Output layer is computed with

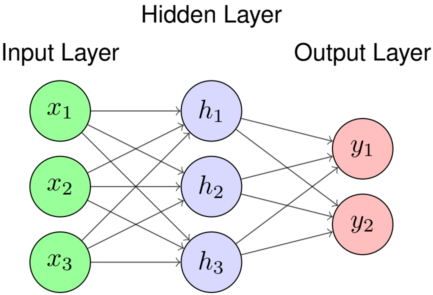
$$\hat{\mathbf{y}} = \boldsymbol{\sigma}(\mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}) \tag{11}$$

- Hence the full network is computed with

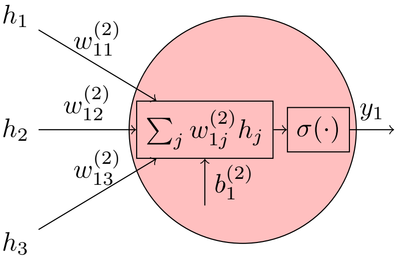
$$\hat{\mathbf{y}} = \underbrace{\boldsymbol{\sigma}(\mathbf{W}^{(2)}\boldsymbol{\sigma}(\mathbf{W}^{(1)}\mathbf{x}+\mathbf{b}^{(1)})+\mathbf{b}^{(2)})}_{\mathbf{f}_{\mathrm{NN}}(\mathbf{x},\boldsymbol{\theta})} \tag{12}$$

where $\boldsymbol{\theta}$ is all the weights and biases.

A Simple Neural Network



A Single Neuron Example:



- Output of neuron y_1 is

$$\hat{y}_1 = \sigma\left(\sum_j w_{1j}^{(2)} h_j + b_1^{(2)}\right) \quad (9)$$

$$\triangleq \sigma(\mathbf{w}_1^{(2)\top}\mathbf{h} + b_1^{(2)}). \quad (10)$$

- Output layer is computed with

$$\hat{\mathbf{y}} = \boldsymbol{\sigma}(\mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}) \quad (11)$$

- Hence the full network is computed with

$$\hat{\mathbf{y}} = \underbrace{\boldsymbol{\sigma}(\mathbf{W}^{(2)}\boldsymbol{\sigma}(\mathbf{W}^{(1)}\mathbf{x}+\mathbf{b}^{(1)})+\mathbf{b}^{(2)})}_{\mathbf{f}_{\mathrm{NN}}(\mathbf{x},\boldsymbol{\theta})} \quad (12)$$

where $\boldsymbol{\theta}$ is all the weights and biases.

- $\sigma(\cdot)$ is called the *activation function*. An example is

$$\sigma(z) = \frac{1}{1+e^{-z}}. \quad (13)$$

Fitting a Neural Network to Data

- Regression with a neural network is then just a matter of minimizing

$$L(\boldsymbol{\theta}, \mathcal{D}) = \frac{1}{2N} \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - \mathbf{f}_{\mathrm{NN}}(\mathbf{x}^{(i)}, \boldsymbol{\theta}) \right\|^2 \tag{14}$$

where $\boldsymbol{\theta} = (\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)})$ is all the weights and biases.

Fitting a Neural Network to Data

- Regression with a neural network is then just a matter of minimizing

$$L(\boldsymbol{\theta}, \mathcal{D}) = \frac{1}{2N} \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - \mathbf{f}_{\mathrm{NN}}(\mathbf{x}^{(i)}, \boldsymbol{\theta}) \right\|^2 \tag{14}$$

where $\boldsymbol{\theta} = (\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)})$ is all the weights and biases.

- We can use gradient descent to optimize the loss numerically

$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathcal{D}) \tag{15}$$

where $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathcal{D}) = (\frac{\partial L(\boldsymbol{\theta}, \mathcal{D})}{\partial \boldsymbol{\theta}})^{\mathsf{T}}$ and α is a step size or *learning rate*.

Fitting a Neural Network to Data

- Regression with a neural network is then just a matter of minimizing

$$L(\boldsymbol{\theta}, \mathcal{D}) = \frac{1}{2N} \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - \mathbf{f}_{\mathrm{NN}}(\mathbf{x}^{(i)}, \boldsymbol{\theta}) \right\|^2 \tag{14}$$

  where $\boldsymbol{\theta} = (\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)})$ is all the weights and biases.
- We can use gradient descent to optimize the loss numerically

$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathcal{D}) \tag{15}$$

  where $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathcal{D}) = (\frac{\partial L(\boldsymbol{\theta}, \mathcal{D})}{\partial \boldsymbol{\theta}})^{\mathsf{T}}$ and α is a step size or *learning rate*.
- $\boldsymbol{\theta}$ can be “columnized” when necessary.
- The next challenge is to determine $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathcal{D})$.

Computing the Gradient

- The final thing required in order to optimize is to compute $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathcal{D})$.
- We will go layer-by-layer, starting with the output layer (the one at the "back"), and make heavy use of the chain rule.

Computing the Gradient

- The loss function can be written as

$$L(\boldsymbol{\theta}, \mathcal{D}) = \frac{1}{2N} \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - \mathbf{f}_{\mathrm{NN}}(\mathbf{x}^{(i)}, \boldsymbol{\theta}) \right\|^2 = \frac{1}{N} \sum_{i=1}^{N} L_i(\boldsymbol{\theta}, (\mathbf{x}^{(i)}, \mathbf{y}^{(i)})), \quad (16)$$

  where the loss for one data point is

$$L_i \left(\boldsymbol{\theta}, (\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) \right) = \frac{1}{2} \left\| \mathbf{y}^{(i)} - \mathbf{f}_{\mathrm{NN}}(\mathbf{x}^{(i)}, \boldsymbol{\theta}) \right\|^2 . \quad (17)$$

Computing the Gradient

- The loss function can be written as

$$L(\boldsymbol{\theta}, \mathcal{D}) = \frac{1}{2N} \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - \mathbf{f}_{\mathrm{NN}}(\mathbf{x}^{(i)}, \boldsymbol{\theta}) \right\|^2 = \frac{1}{N} \sum_{i=1}^{N} L_i(\boldsymbol{\theta}, (\mathbf{x}^{(i)}, \mathbf{y}^{(i)})), \tag{16}$$

  where the loss for one data point is

$$L_i\left(\boldsymbol{\theta}, (\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\right) = \frac{1}{2} \left\| \mathbf{y}^{(i)} - \mathbf{f}_{\mathrm{NN}}(\mathbf{x}^{(i)}, \boldsymbol{\theta}) \right\|^2. \tag{17}$$

- It will also be useful to let $\hat{\mathbf{y}}^{(i)} = \mathbf{f}_{\mathrm{NN}}(\mathbf{x}^{(i)}, \boldsymbol{\theta})$ and to rewrite the norm as a sum,

$$\begin{aligned} L_i\left(\boldsymbol{\theta}, (\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\right) &= \frac{1}{2} \left\| \mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)} \right\|^2, & (18) \\ &= \frac{1}{2} \sum_k \left(y_k^{(i)} - \hat{y}_k^{(i)} \right)^2. & (19) \end{aligned}$$

Computing the Gradient

- The goal is to compute $\frac{\partial L_i}{\partial \boldsymbol{\theta}}$, which means computing $\frac{\partial L_i}{\partial \mathbf{W}^{(1)}}$, $\frac{\partial L_i}{\partial \mathbf{b}^{(1)}}$, $\frac{\partial L_i}{\partial \mathbf{W}^{(2)}}$, and $\frac{\partial L_i}{\partial \mathbf{b}^{(2)}}$.
- In this particular derivation, we will compute the derivatives element-wise.

Parameter	**Derivative**
$w_{jn}^{(1)}$	
$b_j^{(1)}$	
$w_{kj}^{(2)}$	
$b_k^{(2)}$	

Computing the Gradient

- We will compute the gradients element-wise to simplify the notation, beginning with $w_{kj}^{(2)}$. Using the chain rule,

$$\frac{\partial L_i}{\partial w_{kj}^{(2)}} = \frac{\partial L_i}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial w_{kj}^{(2)}}. \tag{20}$$

Computing the Gradient

- We will compute the gradients element-wise to simplify the notation, beginning with $w_{kj}^{(2)}$. Using the chain rule,

$$\frac{\partial L_i}{\partial w_{kj}^{(2)}} = \frac{\partial L_i}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial w_{kj}^{(2)}}. \tag{20}$$

- Recalling that $\hat{y}_k = \sigma\left(\sum_j w_{kj}^{(2)} h_j + b_k^{(2)}\right)$, let

$$a_k = \sum_j w_{kj}^{(2)} h_j + b_k^{(2)} \tag{21}$$

be an *activation*.

Computing the Gradient

- We will compute the gradients element-wise to simplify the notation, beginning with $w_{kj}^{(2)}$. Using the chain rule,

$$\frac{\partial L_i}{\partial w_{kj}^{(2)}} = \frac{\partial L_i}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial w_{kj}^{(2)}}. \tag{20}$$

- Recalling that $\hat{y}_k = \sigma\left(\sum_j w_{kj}^{(2)} h_j + b_k^{(2)}\right)$, let

$$a_k = \sum_j w_{kj}^{(2)} h_j + b_k^{(2)} \tag{21}$$

  be an *activation*.

- The chain rule can then be applied again to yield

$$\frac{\partial L_i}{\partial w_{kj}^{(2)}} = \frac{\partial L_i}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial a_k} \frac{\partial a_k}{\partial w_{kj}^{(2)}}. \tag{22}$$

Computing the Gradient

- Compute each partial derivative in

$$\frac{\partial L_i}{\partial w_{kj}^{(2)}} = \frac{\partial L_i}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial a_k} \frac{\partial a_k}{\partial w_{kj}^{(2)}}. \tag{23}$$

Equation	**Derivative**
$L_i = \frac{1}{2} \sum_k (y_k - \hat{y}_k)^2$	$\frac{\partial L_i}{\partial \hat{y}_k} = \hat{y}_k - y_k$
$\hat{y}_k = \sigma(a_k)$	$\frac{\partial \hat{y}_k}{\partial a_k} = \sigma'(a_k)$
$a_k = \sum_j w_{kj}^{(2)} h_j + b_k^{(2)}$	$\frac{\partial a_k}{\partial w_{kj}^{(2)}} = h_j$

- Therefore,

$$\frac{\partial L_i}{\partial w_{kj}^{(2)}} = (\hat{y}_k - y_k)\sigma'(a_k)h_j. \tag{24}$$

Computing the Gradient

- At this stage, it is useful to introduce

$$\delta_k = \frac{\partial L_i}{\partial a_k} = (\hat{y}_k - y_k)\sigma'(a_k). \tag{25}$$

which leads to

$$\frac{\partial L_i}{\partial w_{kj}^{(2)}} = \delta_k h_j. \tag{26}$$

Computing the Gradient

Parameter	Derivative
$w_{jn}^{(1)}$	
$b_j^{(1)}$	
$w_{kj}^{(2)}$	$\frac{\partial L_i}{\partial w_{kj}^{(2)}} = \delta_k h_j$
$b_k^{(2)}$	

Computing the Gradient

- To compute $\frac{\partial L_i}{\partial b_k^{(2)}}$, the chain rule is applied once again to obtain

$$\frac{\partial L_i}{\partial b_k^{(2)}} = \frac{\partial L_i}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial a_k} \frac{\partial a_k}{\partial b_k^{(2)}}. \tag{27}$$

Computing the Gradient

- To compute $\frac{\partial L_i}{\partial b_k^{(2)}}$, the chain rule is applied once again to obtain

$$\frac{\partial L_i}{\partial b_k^{(2)}} = \frac{\partial L_i}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial a_k} \frac{\partial a_k}{\partial b_k^{(2)}}. \tag{27}$$

- Given that

$$\frac{\partial a_k}{\partial b_k^{(2)}} = \frac{\partial \left(\sum_j w_{kj}^{(2)} h_j + b_k^{(2)} \right)}{\partial b_k^{(2)}} = 1, \tag{28}$$

it follows that

$$\frac{\partial L_i}{\partial b_k^{(2)}} = \delta_k. \tag{29}$$

Computing the Gradient

Parameter	**Derivative**
$w_{jn}^{(1)}$	
$b_j^{(1)}$	
$w_{kj}^{(2)}$	$\frac{\partial L_i}{\partial w_{kj}^{(2)}} = \delta_k h_j$
$b_k^{(2)}$	$\frac{\partial L_i}{\partial b_k^{(2)}} = \delta_k$

Computing the Gradient

- Next up is $\frac{\partial L_i}{\partial w_{jn}^{(1)}}$, for which the chain rule is used to get

$$\frac{\partial L_i}{\partial w_{jn}^{(1)}} = \sum_k \frac{\partial L_i}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial a_k} \frac{\partial a_k}{\partial h_j} \frac{\partial h_j}{\partial w_{jn}^{(1)}}. \tag{30}$$

Computing the Gradient

- Next up is $\frac{\partial L_i}{\partial w_{jn}^{(1)}}$, for which the chain rule is used to get

$$\frac{\partial L_i}{\partial w_{jn}^{(1)}} = \sum_k \frac{\partial L_i}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial a_k} \frac{\partial a_k}{\partial h_j} \frac{\partial h_j}{\partial w_{jn}^{(1)}}. \tag{30}$$

- Recalling that $h_j = \sigma\left(\sum_n w_{jn}^{(1)} x_n + b_j^{(1)}\right)$, let

$$a_j = \sum_n w_{jn}^{(1)} x_n + b_j^{(1)}. \tag{31}$$

Computing the Gradient

- Next up is $\frac{\partial L_i}{\partial w_{jn}^{(1)}}$, for which the chain rule is used to get

$$\frac{\partial L_i}{\partial w_{jn}^{(1)}} = \sum_k \frac{\partial L_i}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial a_k} \frac{\partial a_k}{\partial h_j} \frac{\partial h_j}{\partial w_{jn}^{(1)}}. \tag{30}$$

- Recalling that $h_j = \sigma\left(\sum_n w_{jn}^{(1)} x_n + b_j^{(1)}\right)$, let

$$a_j = \sum_n w_{jn}^{(1)} x_n + b_j^{(1)}. \tag{31}$$

- Once again, the chain rule is used to get

$$\frac{\partial L_i}{\partial w_{jn}^{(1)}} = \sum_k \frac{\partial L_i}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial a_k} \frac{\partial a_k}{\partial h_j} \frac{\partial h_j}{\partial a_j} \frac{\partial a_j}{\partial w_{jn}^{(1)}}. \tag{32}$$

Computing the Gradient

- The missing terms in

$$\frac{\partial L_i}{\partial w_{jn}^{(1)}} = \sum_k \frac{\partial L_i}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial a_k} \frac{\partial a_k}{\partial h_j} \frac{\partial h_j}{\partial a_j} \frac{\partial a_j}{\partial w_{jn}^{(1)}} \tag{33}$$

can then be computed.

Equation	**Derivative**
$a_k = \sum_j w_{kj}^{(2)} h_j + b_k^{(2)}$	$\frac{\partial a_k}{\partial h_j} = w_{kj}^{(2)}$
$h_j = \sigma(a_j)$	$\frac{\partial h_j}{\partial a_j} = \sigma'(a_j)$
$a_j = \sum_n w_{jn}^{(1)} x_n + b_j^{(1)}$	$\frac{\partial a_j}{\partial w_{jn}^{(1)}} = x_n$

- Therefore,

$$\frac{\partial L_i}{\partial w_{jn}^{(1)}} = \sum_k \delta_k w_{kj}^{(2)} \sigma'(a_j) x_n. \tag{34}$$

Computing the Gradient

- It is again useful to introduce

$$\delta_j = \frac{\partial L_i}{\partial a_j} = \sigma'(a_j) \sum_k \delta_k w_{kj}^{(2)}. \quad (35)$$

Computing the Gradient

- It is again useful to introduce

$$\delta_j = \frac{\partial L_i}{\partial a_j} = \sigma'(a_j) \sum_k \delta_k w_{kj}^{(2)}. \tag{35}$$

- This simplifies the gradient to

$$\frac{\partial L_i}{\partial w_{jn}^{(1)}} = \delta_j x_n. \tag{36}$$

Computing the Gradient

Parameter	Derivative
$w^{(1)}_{jn}$	$\frac{\partial L_i}{\partial w^{(1)}_{jn}} = \delta_j x_n$
$b^{(1)}_j$	
$w^{(2)}_{kj}$	$\frac{\partial L_i}{\partial w^{(2)}_{kj}} = \delta_k h_j$
$b^{(2)}_k$	$\frac{\partial L_i}{\partial b^{(2)}_k} = \delta_k$

Computing the Gradient

- The final derivative to compute is

$$\frac{\partial L_i}{\partial b_j^{(1)}} = \frac{\partial L_i}{\partial a_j} \frac{\partial a_j}{\partial b_j^{(1)}}. \tag{37}$$

Computing the Gradient

- The final derivative to compute is

$$\frac{\partial L_i}{\partial b_j^{(1)}} = \frac{\partial L_i}{\partial a_j} \frac{\partial a_j}{\partial b_j^{(1)}}. \tag{37}$$

- As

$$\frac{\partial a_j}{\partial b_j^{(1)}} = \frac{\partial \left(\sum_n w_{jn}^{(1)} x_n + b_j^{(1)} \right)}{\partial b_j^{(1)}} = 1, \tag{38}$$

the final equation is

$$\frac{\partial L_i}{\partial b_j^{(1)}} = \delta_j, \tag{39}$$

Computing the Gradient

Parameter	Derivative
$w_{jn}^{(1)}$	$\frac{\partial L_i}{\partial w_{jn}^{(1)}} = \delta_j x_n$
$b_j^{(1)}$	$\frac{\partial L_i}{\partial b_j^{(1)}} = \delta_j$
$w_{kj}^{(2)}$	$\frac{\partial L_i}{\partial w_{kj}^{(2)}} = \delta_k h_j$
$b_k^{(2)}$	$\frac{\partial L_i}{\partial b_k^{(2)}} = \delta_k$

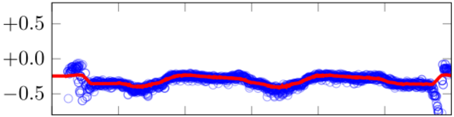


Figure 5: Data (blue) and NN prediction (red) [1]

“Forward” vs. “Reverse” Mode Differentiation

- What we just did is called *backpropagation*. But why?
- Consider a composition of functions which we want to differentiate with respect to its input $\mathbf{x}$,

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}^{(3)}\,\underbrace{(\mathbf{f}^{(2)}(\underbrace{\mathbf{f}^{(1)}(\mathbf{x})}_{\mathbf{h}_1}))}_{\mathbf{h}_2} \tag{40}$$

where $\mathbf{f}(\mathbf{x}) \in \mathbb{R}^D, \mathbf{x} \in \mathbb{R}^d, \mathbf{f}^{(3)}(\mathbf{h}_2) \in \mathbb{R}^D, \mathbf{f}^{(2)}(\mathbf{h}_1) \in \mathbb{R}^{d_2}, \mathbf{f}^{(1)}(\mathbf{x}) \in \mathbb{R}^{d_1}$.

- How do we compute $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ efficiently if we know $\frac{\partial \mathbf{f}^{(3)}}{\partial \mathbf{h}_2}$, $\frac{\partial \mathbf{f}^{(2)}}{\partial \mathbf{h}_1}$, $\frac{\partial \mathbf{f}^{(1)}}{\partial \mathbf{x}}$?

“Forward” vs. “Reverse” Mode Differentiation

Given

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}^{(3)}\underbrace{\left(\mathbf{f}^{(2)}(\underbrace{\mathbf{f}^{(1)}(\mathbf{x})}_{\mathbf{h}_1})\right)}_{\mathbf{h}_2} \tag{41}$$

“Forward” vs. “Reverse” Mode Differentiation

Given

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}^{(3)}\underbrace{(\mathbf{f}^{(2)}(\underbrace{\mathbf{f}^{(1)}(\mathbf{x})}_{\mathbf{h}_1}))}_{\mathbf{h}_2} \tag{41}$$

The Jacobian $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ is given by

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \underbrace{\frac{\partial \mathbf{f}^{(3)}}{\partial \mathbf{h}_2}}_{\mathbb{R}^{D \times d_2}} \underbrace{\frac{\partial \mathbf{f}^{(2)}}{\partial \mathbf{h}_1}}_{\mathbb{R}^{d_2 \times d_1}} \underbrace{\frac{\partial \mathbf{f}^{(1)}}{\partial \mathbf{x}}}_{\mathbb{R}^{d_1 \times d}} .. \tag{42}$$

“Forward” vs. “Reverse” Mode Differentiation

Given

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}^{(3)}\underbrace{(\mathbf{f}^{(2)}(\underbrace{\mathbf{f}^{(1)}(\mathbf{x})}_{\mathbf{h}_1}))}_{\mathbf{h}_2} \tag{41}$$

The Jacobian $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ is given by

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \underbrace{\frac{\partial \mathbf{f}^{(3)}}{\partial \mathbf{h}_2}}_{\mathbb{R}^{D \times d_2}} \underbrace{\frac{\partial \mathbf{f}^{(2)}}{\partial \mathbf{h}_1}}_{\mathbb{R}^{d_2 \times d_1}} \underbrace{\frac{\partial \mathbf{f}^{(1)}}{\partial \mathbf{x}}}_{\mathbb{R}^{d_1 \times d}} .. \tag{42}$$

How do we compute $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ efficiently?

The order of matrix multiplications matters.

“Forward” vs. “Reverse” Mode Differentiation

- Given

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \underbrace{\frac{\partial \mathbf{f}^{(3)}}{\partial \mathbf{h}_2}}_{\mathbb{R}^{D \times d_2}} \underbrace{\frac{\partial \mathbf{f}^{(2)}}{\partial \mathbf{h}_1}}_{\mathbb{R}^{d_2 \times d_1}} \underbrace{\frac{\partial \mathbf{f}^{(1)}}{\partial \mathbf{x}}}_{\mathbb{R}^{d_1 \times d}}.,. \tag{43}$$

[1] $\mathbf{f}^{(1)}$ is the first function applied. So the left direction is indeed the forward one.

“Forward” vs. “Reverse” Mode Differentiation

- Given

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \underbrace{\frac{\partial \mathbf{f}^{(3)}}{\partial \mathbf{h}_2}}_{\mathbb{R}^{D \times d_2}} \underbrace{\frac{\partial \mathbf{f}^{(2)}}{\partial \mathbf{h}_1}}_{\mathbb{R}^{d_2 \times d_1}} \underbrace{\frac{\partial \mathbf{f}^{(1)}}{\partial \mathbf{x}}}_{\mathbb{R}^{d_1 \times d}} .,. \tag{43}$$

- Forward mode would compute

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \underbrace{\color{green}{\frac{\partial \mathbf{f}^{(3)}}{\partial \mathbf{h}_2}} \color{red}{\left(\frac{\partial \mathbf{f}^{(2)}}{\partial \mathbf{h}_1} \frac{\partial \mathbf{f}^{(1)}}{\partial \mathbf{x}} \right)}}_{\longleftarrow\!\!\!\longleftarrow^{1}} \tag{44}$$

---

[1] $\mathbf{f}^{(1)}$ is the first function applied. So the left direction is indeed the forward one.

"Forward" vs. "Reverse" Mode Differentiation

- Given

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \underbrace{\frac{\partial \mathbf{f}^{(3)}}{\partial \mathbf{h}_2}}_{\mathbb{R}^{D \times d_2}} \underbrace{\frac{\partial \mathbf{f}^{(2)}}{\partial \mathbf{h}_1}}_{\mathbb{R}^{d_2 \times d_1}} \underbrace{\frac{\partial \mathbf{f}^{(1)}}{\partial \mathbf{x}}}_{\mathbb{R}^{d_1 \times d}} .,. \tag{43}$$

- Forward mode would compute

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \underbrace{{\color{green}\frac{\partial \mathbf{f}^{(3)}}{\partial \mathbf{h}_2}} {\color{red}\left(\frac{\partial \mathbf{f}^{(2)}}{\partial \mathbf{h}_1} \frac{\partial \mathbf{f}^{(1)}}{\partial \mathbf{x}} \right)}}_{\xleftarrow{\qquad\qquad}^{1}} \tag{44}$$

Cost: ${\color{green}Dd_2d} + {\color{red}(d_2d_1d)} = d(Dd_2 + d_2d_1)$

[1] $\mathbf{f}^{(1)}$ is the first function applied. So the left direction is indeed the forward one.

"Forward" vs. "Reverse" Mode Differentiation

- Given

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \underbrace{\frac{\partial \mathbf{f}^{(3)}}{\partial \mathbf{h}_2}}_{\mathbb{R}^{D\times d_2}} \underbrace{\frac{\partial \mathbf{f}^{(2)}}{\partial \mathbf{h}_1}}_{\mathbb{R}^{d_2\times d_1}} \underbrace{\frac{\partial \mathbf{f}^{(1)}}{\partial \mathbf{x}}}_{\mathbb{R}^{d_1\times d}}.,. \tag{43}$$

- Forward mode would compute

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \underbrace{\frac{\partial \mathbf{f}^{(3)}}{\partial \mathbf{h}_2}\left(\frac{\partial \mathbf{f}^{(2)}}{\partial \mathbf{h}_1}\frac{\partial \mathbf{f}^{(1)}}{\partial \mathbf{x}}\right)}_{\xleftarrow{\hspace{4cm}}^{1}} \tag{44}$$

Cost: $Dd_2d + (d_2d_1d) = d(Dd_2 + d_2d_1)$

- Reverse/backward mode would compute

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \underbrace{\left(\frac{\partial \mathbf{f}^{(3)}}{\partial \mathbf{h}_2}\frac{\partial \mathbf{f}^{(2)}}{\partial \mathbf{h}_1}\right)\frac{\partial \mathbf{f}^{(1)}}{\partial \mathbf{x}}}_{\xrightarrow{\hspace{4cm}}}. \tag{45}$$

Cost: $(Dd_2d_1) + Dd_1d = D(d_2d_1 + d_1d)$.

[1] $\mathbf{f}^{(1)}$ is the first function applied. So the left direction is indeed the forward one.

“Forward” vs. “Reverse” Mode Differentiation

- Given

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \underbrace{\frac{\partial \mathbf{f}^{(3)}}{\partial \mathbf{h}_2}}_{\mathbb{R}^{D\times d_2}} \underbrace{\frac{\partial \mathbf{f}^{(2)}}{\partial \mathbf{h}_1}}_{\mathbb{R}^{d_2\times d_1}} \underbrace{\frac{\partial \mathbf{f}^{(1)}}{\partial \mathbf{x}}}_{\mathbb{R}^{d_1\times d}}.,. \tag{43}$$

- Forward mode would compute

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \underbrace{\color{green}{\frac{\partial \mathbf{f}^{(3)}}{\partial \mathbf{h}_2}} \color{red}{\left(\frac{\partial \mathbf{f}^{(2)}}{\partial \mathbf{h}_1} \frac{\partial \mathbf{f}^{(1)}}{\partial \mathbf{x}} \right)}}_{\xleftarrow{\qquad\qquad\qquad} \text{1}} \tag{44}$$

Cost: $\color{green}{Dd_2d} + \color{red}{(d_2d_1d)} = d(Dd_2 + d_2d_1)$

- Reverse/backward mode would compute

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \underbrace{\color{green}{\left(\frac{\partial \mathbf{f}^{(3)}}{\partial \mathbf{h}_2} \frac{\partial \mathbf{f}^{(2)}}{\partial \mathbf{h}_1} \right)} \color{red}{\frac{\partial \mathbf{f}^{(1)}}{\partial \mathbf{x}}}}_{\xrightarrow{\qquad\qquad\qquad}}. \tag{45}$$

Cost: $\color{green}{(Dd_2d_1)} + \color{red}{Dd_1d} = D(d_2d_1 + d_1d)$. For $D << d$, as with neural nets where the loss $\mathcal{L}(\mathbf{x}|\boldsymbol{\theta}) \in \mathbb{R}$, backward mode is significantly faster.

[1] $\mathbf{f}^{(1)}$ is the first function applied. So the left direction is indeed the forward one.

Neural Networks in Practice

If only it were that easy.

Problem	Solution
Gradients get very complicated	▶ Computational graphs ▶ Automatic differentiation
Overfitting	▶ Regularization ▶ Dropout
Lots of data makes training slow	▶ Stochastic gradient descent ▶ Many other tricks

Automatic Differentiation

- The analytical derivation of backpropagation presented earlier is useful to understand the concept, but is not particularly useful when implementing neural networks.
- There is a need for a method of doing backpropagation without having to analytically compute the derivatives for each loss function, activation function, choice of architecture, etc..
- Automatic differentiation provides a framework for doing just that.

Automatic Differentiation

- Recall the equation for the partial derivative of the loss with respect to the second layer weights,

$$\frac{\partial L_i}{\partial w_{jk}^{(2)}} = \frac{\partial L_i}{\partial \hat{y}_k}\frac{\partial \hat{y}_k}{\partial a_k}\frac{\partial a_k}{\partial w_{jk}^{(2)}}. \tag{46}$$

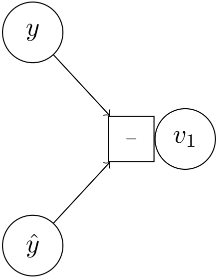
- As an example, we will compute $\frac{\partial L_i}{\partial \hat{y}_k}$ using autodiff.
- To simplify the example even further, we will assume that there is a single output, meaning

$$L_i = \frac{1}{2}(y - \hat{y})^2. \tag{47}$$

Automatic Differentiation

Build the computational graph by decomposing the equation into elementary operations.

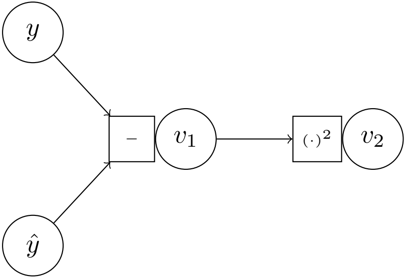
Automatic Differentiation



Build the computational graph by decomposing the equation into elementary operations.

- $v_1 = y - \hat{y}$.

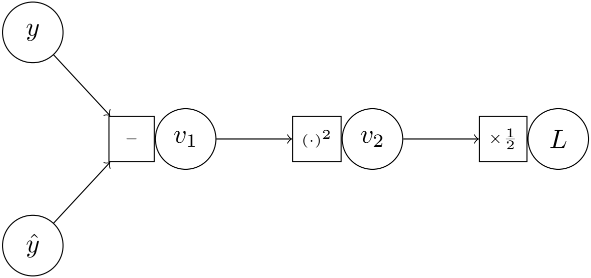
Automatic Differentiation



Build the computational graph by decomposing the equation into elementary operations.

- $v_1 = y - \hat{y}$.
- $v_2 = v_1^2$.

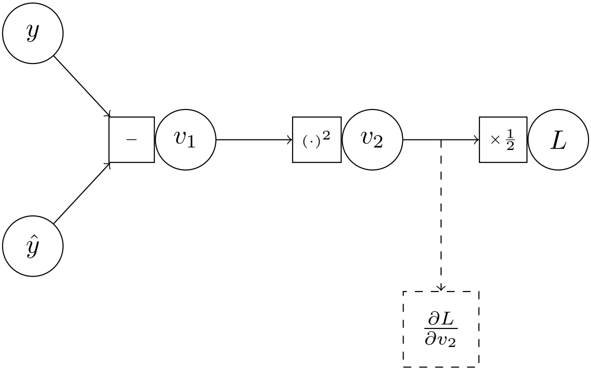
Automatic Differentiation



Build the computational graph by decomposing the equation into elementary operations.

- $v_1 = y - \hat{y}$.
- $v_2 = v_1^2$.
- $L = \frac{1}{2} v_2$.

Automatic Differentiation



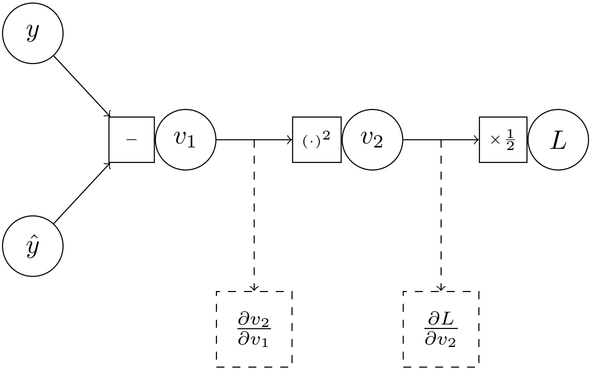
Build the computational graph by decomposing the equation into elementary operations.

- $v_1 = y - \hat{y}$.
- $v_2 = v_1^2$.
- $L = \frac{1}{2}v_2$.

Move backwards through the graph, computing derivatives.

- $\frac{\partial L}{\partial v_2} = \frac{1}{2}$,

Automatic Differentiation



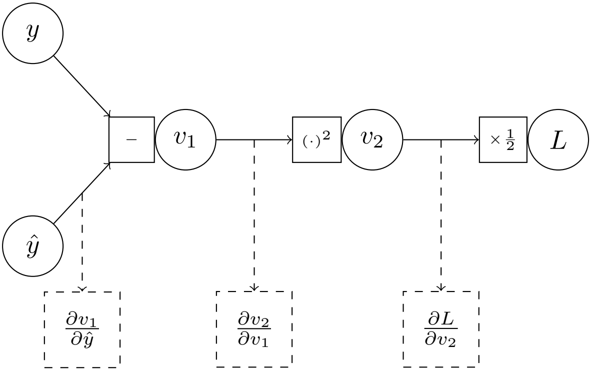
Build the computational graph by decomposing the equation into elementary operations.

- $v_1 = y - \hat{y}$.
- $v_2 = v_1^2$.
- $L = \frac{1}{2} v_2$.

Move backwards through the graph, computing derivatives.

- $\frac{\partial L}{\partial v_2} = \frac{1}{2}$,
- $\frac{\partial v_2}{\partial v_1} = 2v_1$,

Automatic Differentiation



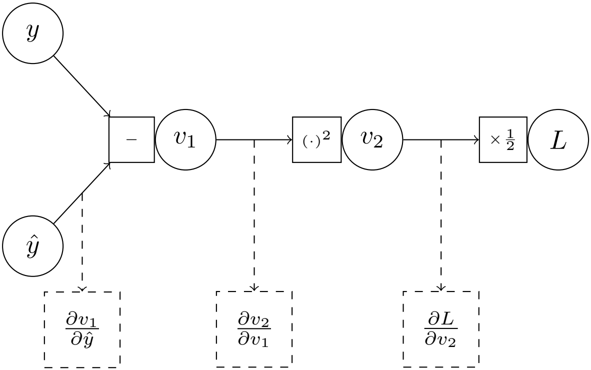
Build the computational graph by decomposing the equation into elementary operations.

- $v_1 = y - \hat{y}$.
- $v_2 = v_1^2$.
- $L = \frac{1}{2} v_2$.

Move backwards through the graph, computing derivatives.

- $\frac{\partial L}{\partial v_2} = \frac{1}{2}$,
- $\frac{\partial v_2}{\partial v_1} = 2v_1$,
- $\frac{\partial v_1}{\partial \hat{y}} = -1$.

Automatic Differentiation



Build the computational graph by decomposing the equation into elementary operations.

- $v_1 = y - \hat{y}$.
- $v_2 = v_1^2$.
- $L = \frac{1}{2} v_2$.

Move backwards through the graph, computing derivatives.

- $\frac{\partial L}{\partial v_2} = \frac{1}{2}$,
- $\frac{\partial v_2}{\partial v_1} = 2v_1$,
- $\frac{\partial v_1}{\partial \hat{y}} = -1$.

- The final equation is found by multiplying,

$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial v_2}\frac{\partial v_2}{\partial v_1}\frac{\partial v_1}{\partial \hat{y}} = \tfrac{1}{2} \times 2v_1 \times -1 = -v_1. \tag{48}$$

Overfitting

- Increasing the complexity of our model makes it more accurate... right?

Overfitting

- Increasing the complexity of our model makes it more accurate... right?
- Well.. sort of.

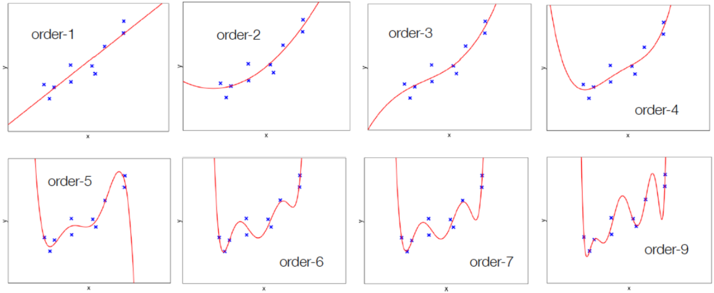


Figure 6: Fitting polynomials of various orders to a 2D dataset. Taken from `https://cs.mcgill.ca/~wlh/comp451/files/comp451_chap10.pdf`

Regularization

- Model needs to **generalize** well.

Regularization

- Model needs to **generalize** well.
- Regularization penalizes complexity in the model.

Regularization

- Model needs to **generalize** well.
- Regularization penalizes complexity in the model.

L2 (Tikhonov) regularization

Penalize squared norm of model parameters.

$$L_{\mathrm{reg}}(\boldsymbol{\theta}, \mathcal{D}) = L(\boldsymbol{\theta}, \mathcal{D}) + \lambda \sum_{i=1}^{n_{\mathrm{param}}} \theta_i^2 \tag{49}$$

L1 (Lasso) regularization

Penalize L1 norm of model parameters.

$$L_{\mathrm{reg}}(\boldsymbol{\theta}, \mathcal{D}) = L(\boldsymbol{\theta}, \mathcal{D}) + \lambda \sum_{i=1}^{n_{\mathrm{param}}} |\theta_i| \tag{50}$$

- L2 regression tends to give parameters with smaller values so that small change in input gives small change in output. L1 tends to drive some parameters to zero, getting rid of useless connections.

Activation functions

- The sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$ is not the only choice.
- Logistic and tanh functions saturate at high and low values which can make gradient-based training difficult. [p. 195][4]

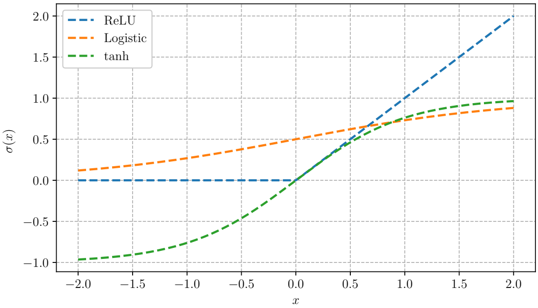


Figure 7: Examples of activation functions

Sigmoid:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{51}$$

tanh:

$$\sigma(x) = \tanh(x) \tag{52}$$

Rectified Linear Unit (ReLU):

$$\sigma(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{53}$$

Maximum Likelihood

Recall that for linear regression, the error is given by the mean squared error,

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} (y^{(i)} - f_{\mathrm{NN}}(\mathbf{x}^{(i)}, \boldsymbol{\theta}))^2. \tag{54}$$

But, if $f_{\mathrm{NN}}(\mathbf{x}^{(i)}, \boldsymbol{\theta}) \in [0, 1]$ estimates a probability for classification, a common approach is maximum likelihood estimation. For binary classification,

$$\begin{aligned}
\boldsymbol{\theta}^* &= \arg\max_{\boldsymbol{\theta}} p(y^{(1)}, \ldots, y^{(N)} | \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}, \boldsymbol{\theta}) && (55)\\
&= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{N} p(y_i | f_{\mathrm{NN}}(\mathbf{x}^{(i)}, \boldsymbol{\theta})) && (56)\\
&= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{N} f_{\mathrm{NN}}(\mathbf{x}^{(i)}, \boldsymbol{\theta})^{y^{(i)}} (1 - f_{\mathrm{NN}}(\mathbf{x}^{(i)}, \boldsymbol{\theta}))^{1-y^{(i)}}. && (57)
\end{aligned}$$

Eq. (57) makes sense if you consider edge cases. For example, given a point $\mathbf{x}^{(1)}$ with $y^{(1)} = 1$ but the model predicts $\mathbf{f}(\mathbf{x}^{(1)} | \boldsymbol{\theta}) = 0.1$. Then likelihood of $\mathbf{x}^{(1)}, y^{(1)}$ is $p(y^{(1)} | f(\mathbf{x}^{(1)}, \boldsymbol{\theta})) = 0.1^1 (1 - 0.1)^0 = 0.1$.

Stochastic Gradient Descent

Gradient Descent

The parameters are updated using

$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathcal{D}). \tag{58}$$

- Stable, but slow.

Stochastic Gradient Descent

Gradient Descent

The parameters are updated using

$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathcal{D}). \tag{58}$$

- Stable, but slow.

Stochastic Gradient Descent

The parameters are updated using

$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, (x^{(i)}, y^{(i)})), \qquad \underbrace{i = 1, \ldots, N}_{\text{an } \textit{epoch}}. \tag{59}$$

- Fast, but unstable.

Mini-Batch Gradient Descent

Mini-Batch Gradient Descent

The parameters are updated using

$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathcal{B}^{(i)}), \qquad \underbrace{i = 1, \ldots, M}_{\text{an } \textit{epoch}}, \tag{60}$$

where the dataset $\mathcal{D}$ is partitioned into M mini-batches, $\mathcal{D} = \{\mathcal{B}^{(1)}, \ldots, \mathcal{B}^{M}\}$, with each mini batch containing K data samples $(\mathbf{x}, y)$.

Stochastic Gradient Descent

To obtain good convergence, several parameters require tuning.

- **Batch Size**: Trade-off between speed and stability.
- **Learning Rate**: Trade-off between speed and stability.
- **Number of Epochs**: Training for too long may result in overfitting (early stopping).

Extensions and Variations of SGD

- Learning rate scheduling: lowering the learning rate as the algorithm approaches the solution.
- Momentum: Update is a linear combination of the gradient and the previous update,

$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathcal{D}) + \eta \Delta \boldsymbol{\theta}, \tag{61}$$

  where $\Delta \boldsymbol{\theta} = \boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1}$. Reduces oscillations, biases algorithm to keep moving in the same direction.
- Adaptive learning rates: Automatically adjust learning rate for each parameter through training.

MNIST Case Study

- MNIST (Modified National Institute of Standards and Technology database)[2]contains handwritten digits, 28 $\times$ 28 pixels, 60 000 training examples and 10 000 testing examples.
- Classic “easy” benchmark dataset where we want to recognize the digit on the image.

Figure 8: Examples of handwritten digits from the MNIST dataset[3]

[1] http://yann.lecun.com/exdb/mnist/

[2] https://en.wikipedia.org/wiki/MNIST_database

MNIST Case Study

- By “flattening” the input from a 28×28 matrix to a 784×1 matrix, a simple neural network can be used to classify the images.
- We will train a network with a single hidden layer with 128 nodes.
- The hidden layer uses a ReLU activation function, while the output layer uses a softmax activation function.
- Mini-batch SGD with adaptive learning rates is used to train the model.
- The effect of the learning rate, batch size and number of epochs is investigated.

MNIST Case Study: Learning Rate

- **Small learning rate**: Slow convergence.
- **Large learning rate**: Unstable training.

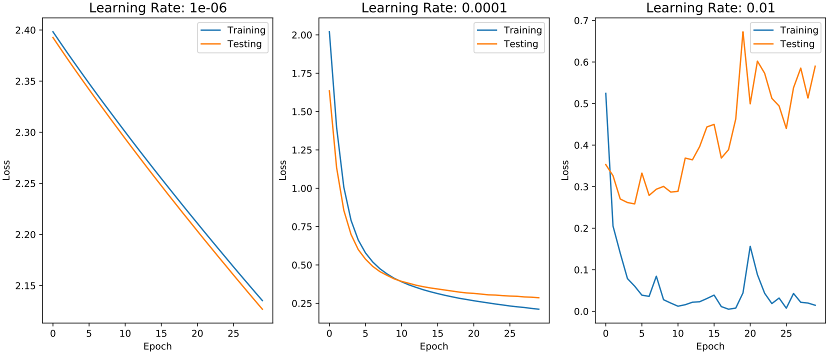


Figure 9: Effect of learning rate on neural network training, training for 30 epochs with a batch size of 64.

MNIST Case Study: Batch Size

- **Small batch size**: Unstable training and overfitting.
- **Large batch size**: Slow convergence.

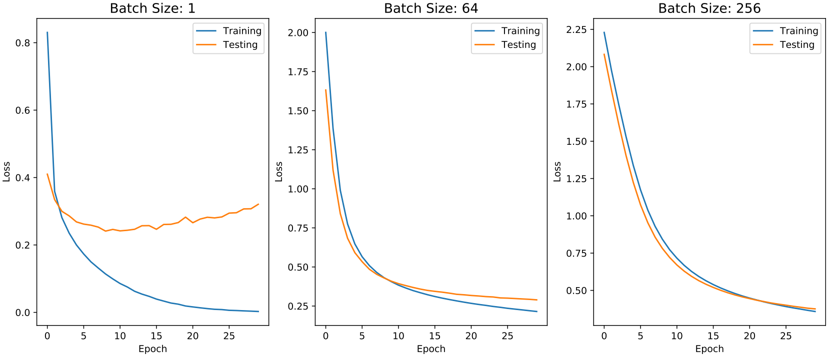


Figure 10: Effect of batch size on neural network training, training for 30 epochs with a learning rate of 10^{-4}.

MNIST Case Study: Number of Epochs

- **Too few epochs**: Model is underfit and performance could still be improved.
- **Too many epochs**: Model is overfit..

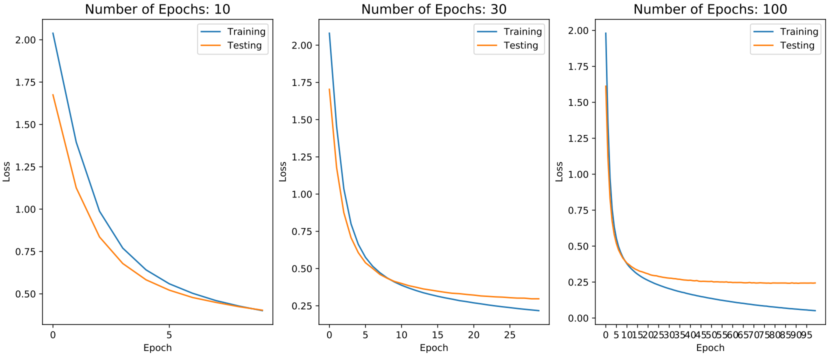


Figure 11: Effect of batch size on neural network training, training with a batch size of 64 with a learning rate of 10^{-4}.

MNIST Case Study

- Using the parameters which yielded the best results, an accuracy of 91.8%.
- Best practice would be to perform a **grid search** with **cross-validation**.

Convolutional Neural Network

In the MNIST neural network example, the image was flattened to a 1D column matrix and fed into the feedforward network. Problems with this:

1. Number of parameters blows up quickly for high resolution images.
2. Lose spatial information.

A convolutional network addresses these problems.

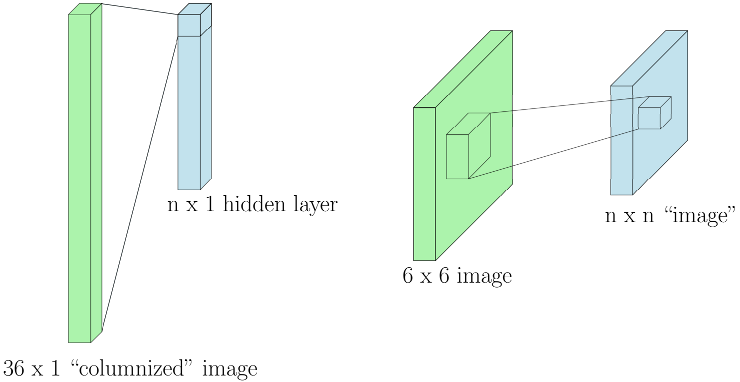


Figure 12: (left) "Plain vanilla" layer. (right) Convolutional layer.

A Single Convolutional Layer

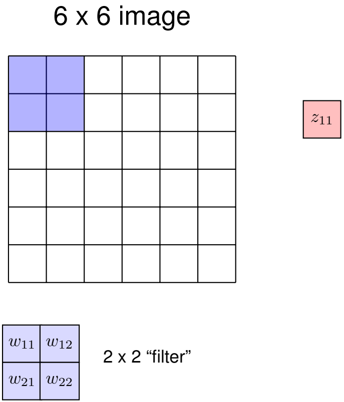


Figure 13: CNN Diagram.

$$z_{ij} = \sum_m \sum_n w_{mn} x_{i+m,j+n} \tag{62}$$

A Single Convolutional Layer

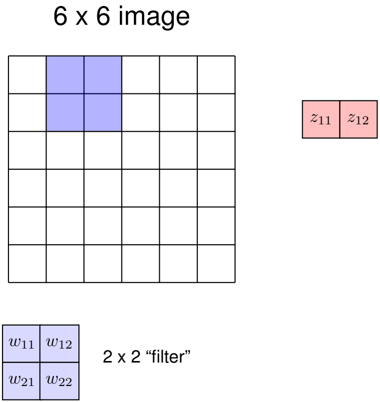


Figure 13: CNN Diagram.

$$z_{ij} = \sum_m \sum_n w_{mn} x_{i+m,j+n} \tag{62}$$

A Single Convolutional Layer

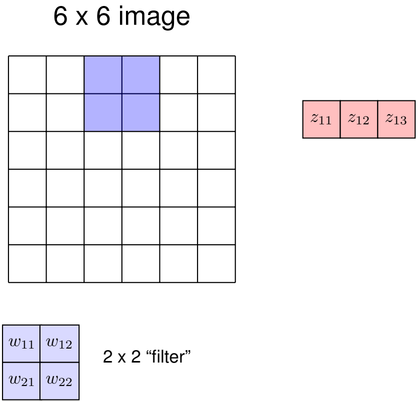


Figure 13: CNN Diagram.

$$z_{ij} = \sum_m \sum_n w_{mn} x_{i+m,j+n} \tag{62}$$

A Single Convolutional Layer

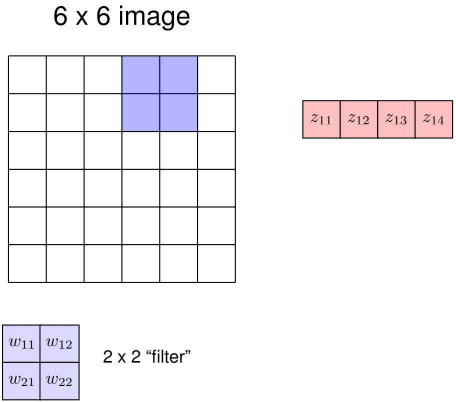


Figure 13: CNN Diagram.

$$z_{ij} = \sum_m \sum_n w_{mn} x_{i+m,j+n} \tag{62}$$

A Single Convolutional Layer

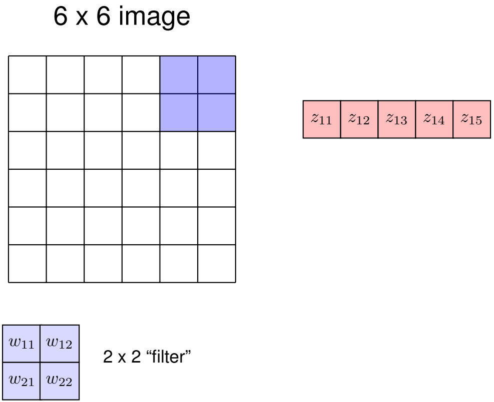


Figure 13: CNN Diagram.

$$z_{ij} = \sum_m \sum_n w_{mn} x_{i+m,j+n} \tag{62}$$

A Single Convolutional Layer

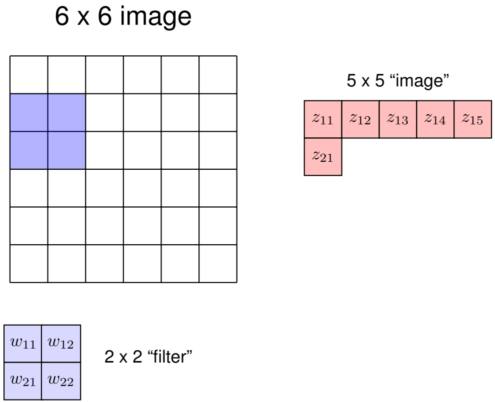


Figure 13: CNN Diagram.

$$z_{ij} = \sum_m \sum_n w_{mn} x_{i+m,j+n} \tag{62}$$

A Single Convolutional Layer

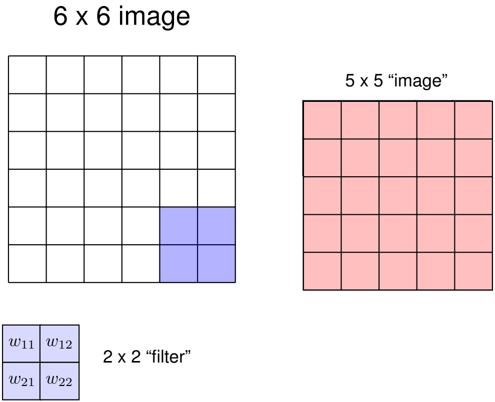


Figure 13: CNN Diagram.

$$z_{ij} = \sum_m \sum_n w_{mn} x_{i+m,j+n} \tag{62}$$

Typical CNN Setup

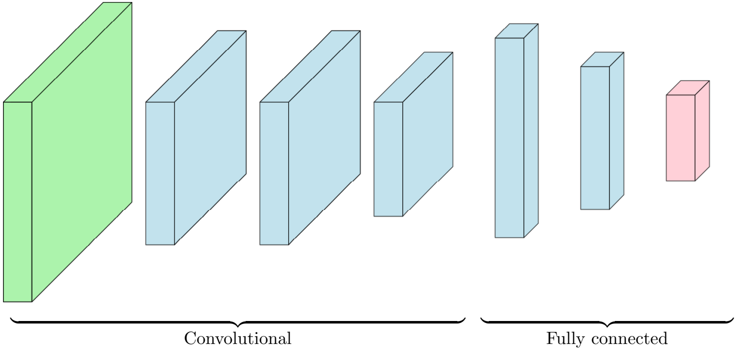


Figure 14: Typical CNN setup, consisting of convolutional layers followed by fully-connected layers.

MNIST Case Study Revisited

- A CNN can also be used to perform handwritten digit recognition.
- The chosen architecture uses 16 3×3 filters and a single fully connected layer with 20 nodes.
- The fully connected network from earlier had 101770 parameters, while this CNN has 54470 parameters.

MNIST Case Study Revisited

- The CNN achieves an accuracy of 94.0% compared to 91.8% for the NN.

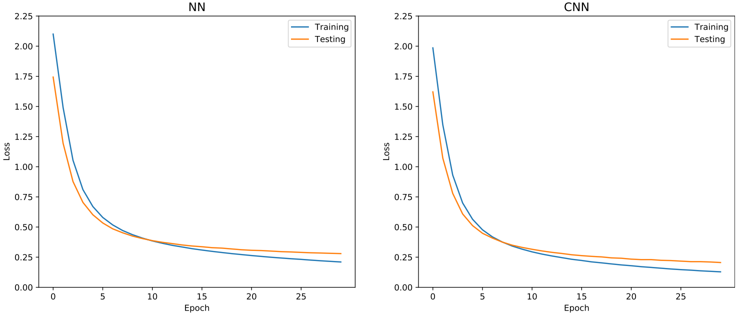


Figure 15: Learning curves for simple neural network (NN) and CNNs.

Recurrent Neural Networks

- A feedforward neural network is well suited for tasks where the inputs are of **fixed lengths** and are **unordered**.

Recurrent Neural Networks

- A feedforward neural network is well suited for tasks where the inputs are of **fixed lengths** and are **unordered**.
- What if the problem involves inputs of variable lengths in which the order matters. For example, machine translation.

Recurrent Neural Networks

- A feedforward neural network is well suited for tasks where the inputs are of **fixed lengths** and are **unordered**.
- What if the problem involves inputs of variable lengths in which the order matters. For example, machine translation.
- A recurrent neural network (RNN) is ideally suited for this problem as it **shares weights** between inputs.

Recurrent Neural Networks

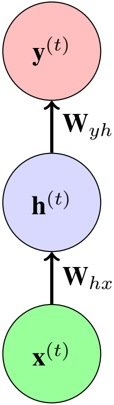


Figure 16: RNN Diagram

Recurrent Neural Networks

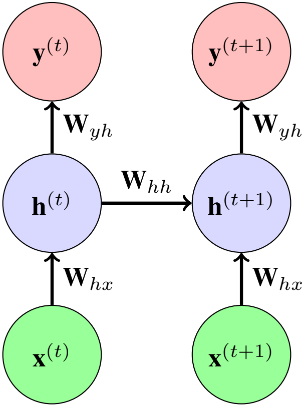


Figure 16: RNN Diagram

Recurrent Neural Networks

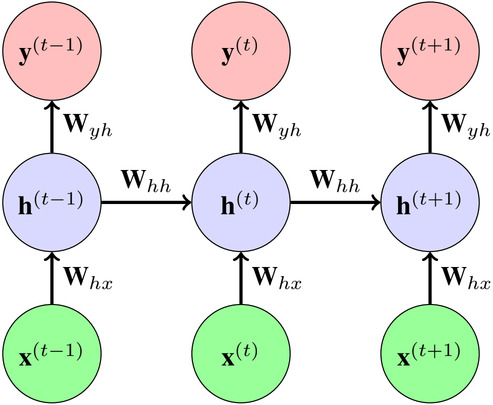


Figure 16: RNN Diagram

Recurrent Neural Networks

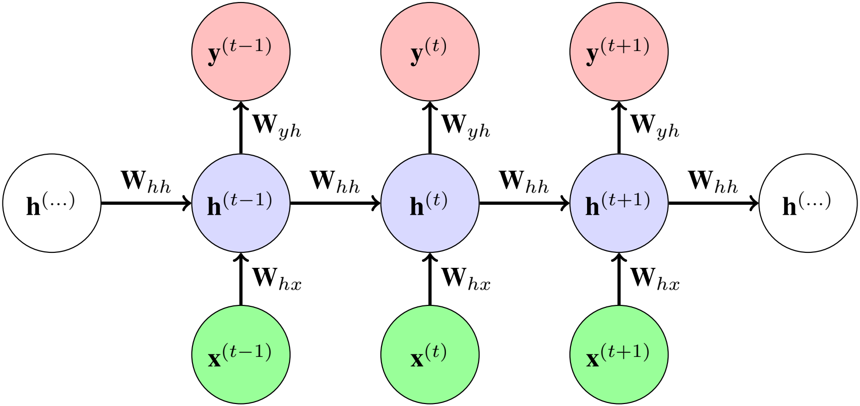


Figure 16: RNN Diagram

Recurrent Neural Networks

Forward Propagation

The hidden unit at time t is computed using

$$\mathbf{h}^{(t)} = \sigma(\mathbf{W}_{hx}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)}). \tag{63}$$

Note this requires the initialization of $\mathbf{h}^{(0)}$. The output at each time step is then computed using.

$$\hat{\mathbf{y}}^{(t)} = \mathbf{W}_{yh}\mathbf{h}^{(t)}. \tag{64}$$

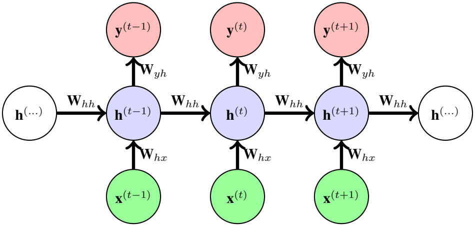


Figure 17: RNN diagram.

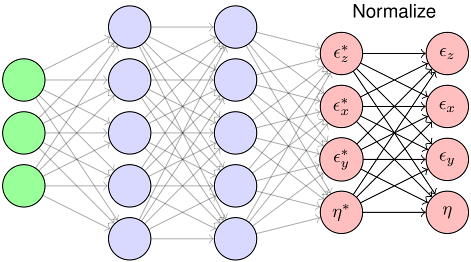
Attitude output from NNs: Quaternions

- Recall that a unit quaternion $\mathbf{q} = [\boldsymbol{\epsilon}^\mathsf{T} \eta]^\mathsf{T}$ can be used to represent attitude.
- However, quaternions must satisfy the following **unit-norm constraint**,

$$\mathbf{q}^\mathsf{T}\mathbf{q} = 1. \tag{65}$$

- We can normalized an un-normalized quaternion $\mathbf{q}^*$ with

$$\mathbf{q} = \frac{\mathbf{q}^*}{\sqrt{\mathbf{q}^{*\mathsf{T}}\mathbf{q}^*}} \triangleq \boldsymbol{\sigma}_{\text{norm}}(\mathbf{q}^*) \tag{66}$$



Attitude output from NNs: Quaternions

- Our “neural network” can be whatever we want! As long as we have a well-defined derivative.
- Thankfully,

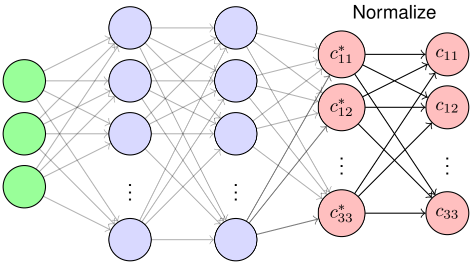
$$\frac{\partial \boldsymbol{\sigma}_{\mathrm{norm}}}{\partial \mathbf{q}^*} = \frac{{\mathbf{q}^*}^\mathsf{T}}{\sqrt{{\mathbf{q}^*}^\mathsf{T}\mathbf{q}^*}}. \tag{67}$$

Attitude output from NNs: DCMs $SO(3)$

- Neural Networks can also predict DCMs $\mathbf{C} \in SO(3)$ directly.

$$\mathbf{C}^{\mathsf{T}}\mathbf{C} = \mathbf{1} \tag{68}$$

- DCMs must also satisfy an orthonormality constraint
- We can take the same philosophy and just normalized an un-normalized DCM (just a matrix) $\mathbf{C}^* \in \mathbb{R}^{3\times 3}$



Attitude output from NNs: DCMs $SO(3)$

- In [5], it is shown that the best option for training is to use an SVD to normalize a DCM.
- Let $\mathbf{C}^* = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\mathsf{T}$ be a SVD. A DCM can be obtained with

$$\begin{aligned} \mathbf{C} &= \mathbf{U}\tilde{\boldsymbol{\Sigma}}\mathbf{V}^\mathsf{T} \in SO(3), \quad \text{where} \quad \tilde{\boldsymbol{\Sigma}} = \operatorname{diag}(1, \ldots, 1, \det(\mathbf{U}\mathbf{V}^\mathsf{T})) && (69) \\ &\triangleq \boldsymbol{\sigma}_{\mathrm{SVD}}(\mathbf{C}^*) && (70) \end{aligned}$$

- It turns out that

$$\boldsymbol{\sigma}_{\mathrm{SVD}}(\mathbf{C}^*) = \underset{\mathbf{C} \in SO(3)}{\arg\min} \|\mathbf{C} - \mathbf{C}^*\|_F^2 \quad (71)$$

  and that the derivative of $\boldsymbol{\sigma}_{\mathrm{SVD}}(\mathbf{C}^*)$ is also well defined! (See [5]).

References

[1] W. Zhao, A. Goudar, J. Panerati, and A. P. Schoellig, "Learning-based Bias Correction for Ultra-wideband Localization of Resource-constrained Mobile Robots,", no. ii, 2020.
[2] M. Luiza, C. Vianna, E. Goubault, and S. Putot, "Neural Network Based Model Predictive Control for an Autonomous Vehicle,",
[3] M. Brossard, A. Barrau, and S. Bonnabel, "AI-IMU Dead-Reckoning," *IEEE Transactions on Intelligent Vehicles*, pp. 1–1, 2020. arXiv: 1904.06064. [Online]. Available: https://github.com/mbrossar/ai-imu-dr.
[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.
[5] J. Levinson, C. Esteves, K. Chen, N. Snavely, A. Kanazawa, A. Rostamizadeh, and A. Makadia, "An Analysis of SVD for Deep Rotation Estimation,", no. 3, pp. 1–18, 2020. arXiv: 2006.14616. [Online]. Available: http://arxiv.org/abs/2006.14616.