Today

- Forward propagation
- Backward propagation
- Deep learning
Motivation Examples

What can I help you with?

Download from
Dreamstime.com

Images and photos by Dreamstime.com

URBAN

Lee Blackwood (Dreamstime.com)
Are you excited about deep learning?
Limitations of linear classifiers

- Linear classifiers (e.g., logistic regression) classify inputs based on linear combinations of features $x_i$
- Many decisions involve non-linear functions of the input
- Canonical example: do 2 input elements have the same value?

$$\begin{array}{c|c|c|c}
0,0 & 0,1 & 1,0 & 1,1 \\
\hline
\text{output} = 0 & \text{output} = 1 & \text{output} = 1 & \text{output} = 0 \\
\end{array}$$

- The positive and negative cases cannot be separated by a plane
- What can we do?
How to construct nonlinear classifiers?

- Would like to construct non-linear discriminative classifiers that utilize functions of input variables
- Add large number of extra functions
  - If these functions are fixed (Gaussian, sigmoid, polynomial basis functions), then optimization still involves linear combinations of (fixed functions of) the inputs
  - Or we can make these functions depend on additional parameters → need an efficient method of training extra parameters
Neural Networks

- Many machine learning methods inspired by biology, brains
- Our brains contain $\sim 10^{11}$ neurons, each of which communicates to $\sim 10^4$ other neurons
- **Multi-layer perceptron**, or **neural network**, is a popular supervised approach
- Defines extra functions of the inputs (**hidden features**), computed by neurons
- Artificial neurons called **units**
- Network output is a linear combination of hidden units
Neural network architecture

- Network with one layer of four hidden units:

- Each unit computes its value based on linear combination of values of units that point into it.

- Can add more layers of hidden units: deeper hidden unit response depends on earlier hiddens.

![Diagram of neural network](image-url)
Neural Networks

- We only need to know two algorithms
  - Forward pass: performs inference
  - Backward pass: performs learning
What does the network compute?

- Output of network can be written as (with $k$ indexing the two output units):

$$h_j(x) = f(w_{j0} + \sum_{i=1}^{D} x_i v_{ji})$$

$$o_k(x) = g(w_{k0} + \sum_{j=1}^{J} h_j(x) w_{kj})$$

- Network with non-linear activation function $f()$ is a universal approximator (esp. with increasing $J$)

- Standard $f$: sigmoid/logistic, or tanh, or rectified linear (relu)

$$tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)} \quad relu(z) = \max(0, z)$$
Example application

- Consider trying to classify image of handwritten digit: 32x32 pixels

- Single output units – it is a 4 (one vs. all)?

- Use the sigmoid output function:

\[
o_k(x) = \frac{1}{1 + \exp(-z_k)}
\]

\[
z_k = w_{k0} + \sum_{j=1}^{J} h_j(x)w_{kj}
\]

- What do I recover if \( h_j(x) = x_j \)?

- How can we train the network, that is, adjust all the parameters \( \mathbf{w} \)?

- If we have trained the network, how can we do inference?
Training multi-layer networks: back-propagation

- Use gradient descent to learn the weights
- **Back-propagation**: an efficient method for computing gradients needed to perform gradient-based optimization of the weights in a multi-layer network
- Loop until convergence:
  - for each example $n$
    1. Given input $x^{(n)}$, propagate activity forward ($x^{(n)} \rightarrow h^{(n)} \rightarrow o^{(n)}$)
    2. Propagate gradients backward
    3. Update each weight (via gradient descent)
- Given any error function $E$, activation functions $g()$ and $f()$, just need to derive gradients
Key idea behind backpropagation

- We don’t have targets for a hidden unit, but we can compute how fast the error changes as we change its activity
  - Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities
  - Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined
  - We can compute error derivatives for all the hidden units efficiently
  - Once we have the error derivatives for the hidden activities, it's easy to get the error derivatives for the weights going into a hidden unit

- This is just the chain rule!
Computing gradient: single layer network

Error gradients for single layer network:

\[ \frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial z_k} \frac{\partial z_k}{\partial w_{ki}} \]

• Error gradient is computable for any continuous activation function \( g() \), and any continuous error function.
Gradient descent for single layer network

- Assuming the error function is mean-squared error (MSE), on a single training example $n$, we have

$$\frac{\partial E}{\partial o_k^{(n)}} = o_k^{(n)} - t_k^{(n)}$$

Using logistic activations

$$o_k^{(n)} = g(z_k^{(n)}) = (1 + \exp(z_k^{(n)}))^{-1}$$

$$\frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} = o_k^{(n)}(1 - o_k^{(n)})$$

- The error gradient is then:

$$\frac{\partial E}{\partial w_{ki}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{ki}} = \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)}(1 - o_k^{(n)}) x_i^{(n)}$$

- The gradient descent update rule is given by:

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)}(1 - o_k^{(n)}) x_i^{(n)}$$
Multi-layer neural network

- **Output Layer**
  - $O_k$: Output of unit $k$
  - $Z_k$: Net input to output unit $k$
  - $w_{kj}$: Weight from hidden $j$ to output $k$

- **Hidden Layer**
  - $h_j$: Output of hidden unit $j$
  - $U_j$: Net input to unit $j$
  - $v_{ji}$: Weight from input $i$ to output $j$

- **Input Layer**
  - $x_i$: Input unit $i$
Back-propagation: sketch on one training case

- Convert discrepancy between each output and its target value into an error derivative
  \[ E = \frac{1}{2} \sum_k (o_k - t_k)^2; \quad \frac{\partial E}{\partial o_k} = o_k - t_k \]

- Compute error derivatives in each hidden layer from error derivatives in layer above. [assign blame for error at \( k \) to each unit \( j \) according to its influence on \( k \) (depends on \( w_{kj} \))]

- Use error derivatives w.r.t. activities to get error derivatives w.r.t. the weights.
Gradient descent for multi-layer network

- The output weight gradients for a multi-layer network are the same as for a single layer network

\[ \frac{\partial E}{\partial w_{kj}} = N \sum_{n=1}^{N} \frac{\partial E}{\partial o_{k}^{(n)}} \frac{\partial o_{k}^{(n)}}{\partial z_{k}^{(n)}} \frac{\partial z_{k}^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_{k}^{(n)} h_{j}^{(n)} \]

where \( \delta_{k} \) is the error w.r.t. the net input for unit \( k \)

- Hidden weight gradients are then computed via back-prop:

\[ \frac{\partial E}{\partial h_{j}^{(n)}} = \sum_{k} \frac{\partial E}{\partial o_{k}^{(n)}} \frac{\partial o_{k}^{(n)}}{\partial z_{k}^{(n)}} \frac{\partial z_{k}^{(n)}}{\partial h_{j}^{(n)}} = \sum_{k} \delta_{k}^{(n)} w_{kj} \]

\[ \frac{\partial E}{\partial v_{ji}} = \sum_{n=1}^{N} \frac{\partial E}{\partial h_{j}^{(n)}} \frac{\partial h_{j}^{(n)}}{\partial u_{j}^{(n)}} \frac{\partial u_{j}^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \left( \sum_{k} \delta_{k}^{(n)} w_{kj} \right) f'(u_{j}^{(n)}) x_{i}^{(n)} = \sum_{n=1}^{N} \tilde{\delta}_{j}^{(n)} x_{i}^{n} \]
Choosing activation and cost functions

- When using a neural network as a function approximator (regressor) sigmoid activation and MSE as loss function work well.

- For classification, if it is a binary (2-class) problem, then cross-entropy error function often does better (as we saw with logistic regression)

\[
E = - \sum_{n=1}^{N} t^{(n)} \log o^{(n)} + (1 - t^{(n)}) \log(1 - o^{(n)})
\]

\[
o^{(n)} = \frac{1}{1 + \exp(-z^{(n)})}
\]

- We can then compute via the chain rule

\[
\frac{\partial E}{\partial o} = o - t
\]

\[
\frac{\partial o}{\partial z} = o(1 - o)
\]

\[
\frac{\partial E}{\partial z} = \frac{\partial E}{\partial o} \frac{\partial o}{\partial z} = (o - t) o(1 - o)
\]
Multi-class classification

For multi-class classification problems, use the softmax activation

\[ E = - \sum_n \sum_k t_k^{(n)} \log o_k^{(n)} \]

\[ o_k^{(n)} = \frac{\exp(z_k^{(n)})}{\sum_j \exp(z_j^{(n)})} \]

And the derivatives become

\[ \frac{\partial o_k}{\partial z_k} = o_k (1 - o_k) \]

\[ \frac{\partial E}{\partial z_k} = \sum_j \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial z_k} = (o_k - t_k) o_k (1 - o_k) \]
Now trying to classify image of handwritten digit: 32x32 pixels

10 output units, 1 per digit

Use the softmax function:

\[ o_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)} \]

\[ z_k = w_{k0} + \sum_{j=1}^{J} h_j(x)w_{kj} \]

What is \( J \)?
Ways to use weight derivatives

- How often to update
  - after a full sweep through the training data
    \[ w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} (o_{k}^{(n)} - t_{k}^{(n)})o_{k}^{(n)}(1 - o_{k}^{(n)})x_{i}^{(n)} \]
  - after each training case
  - after a mini-batch of training cases

- How much to update
  - Use a fixed learning rate
  - Adapt the learning rate
  - Add momentum
    \[ w_{ki} \leftarrow w_{ki} - \nu \\
    \nu \leftarrow \gamma \nu + \eta \frac{\partial E}{\partial w_{ki}} \]
Deep Neural Networks

- We only need to know two algorithms
  - Forward pass: performs inference
  - Backward pass: performs learning

- Neural nets are now called deep learning. Why?
Supervised Learning: Examples

Classification

Supervised Deep Learning

Classification

[Picture from M. Ranzato]
Deep learning uses **composite of simple functions** (e.g., ReLU, sigmoid, tanh, max) to create complex non-linear functions.

Note: a composite of linear functions is linear!

Example: 2 layer NNet (now matrix and vector form!)

\[
x \xrightarrow{\text{max}(0, W_1^T x + b_1)} h^1 \xrightarrow{\text{max}(0, W_2^T h^1 + b_2)} h^2 \xrightarrow{W_3^T h^2 + b_3} y
\]

- $x$ is the input
- $y$ is the output (what we want to predict)
- $h^i$ is the $i$-th hidden layer
- $W^i$ are the parameters of the $i$-th layer
Evaluating the Function

- Assume we have learned the weights and we want to do inference.

- **Forward Propagation:** compute the output given the input.

\[
\begin{align*}
    x & \rightarrow \max(0, W_1^T x + b^1) \rightarrow \max(0, W_2^T h^1 + b^2) \rightarrow W_3^T h^2 + b^3 \rightarrow y
\end{align*}
\]

- **Fully connected layer:** Each hidden unit takes as input all the units from the previous layer.

- The non-linearity is called a ReLU (rectified linear unit), with \( x \in \mathbb{R}^D \), \( b^i \in \mathbb{R}^{N_i} \) the biases and \( W^i \in \mathbb{R}^{N_i \times N_{i-1}} \) the weights.

- Do it in a compositional way,

\[
h^1 = \max(0, W^1 x + b^1)
\]
Evaluating the Function

- Assume we have learned the weights and we want to do **inference**
- **Forward Propagation:** compute the output given the input

\[ x \rightarrow \text{max}(0, W_1^T x + b^1) \rightarrow \text{max}(0, W_2^T h^1 + b^2) \rightarrow W_3^T h^2 + b^3 \rightarrow y \]

- **Fully connected layer:** Each hidden unit takes as input all the units from the previous layer
- The non-linearity is called a ReLU (rectified linear unit), with \( x \in \mathbb{R}^D \), \( b^i \in \mathbb{R}^{N_i} \) the biases and \( W^i \in \mathbb{R}^{N_i \times N_{i-1}} \) the weights
- Do it in a compositional way

\[
\begin{align*}
    h^1 &= \text{max}(0, W^1 x + b^1) \\
    h^2 &= \text{max}(0, W^2 h^1 + b^2)
\end{align*}
\]
Evaluating the Function

- Assume we have learned the weights and we want to do inference.

- **Forward Propagation**: compute the output given the input.

  \[
  x \rightarrow \text{max}(0, W_1^T x + b^1) \rightarrow \text{max}(0, W_2^T h^1 + b^2) \rightarrow \text{max}(0, W_3^T h^2 + b^3) \rightarrow y
  \]

- **Fully connected layer**: Each hidden unit takes as input all the units from the previous layer.

  - The non-linearity is called a ReLU (rectified linear unit), with \( x \in \mathbb{R}^D \), \( b^i \in \mathbb{R}^{N_i} \) the biases and \( W^i \in \mathbb{R}^{N_i \times N_{i-1}} \) the weights.

- Do it in a compositional way:

  \[
  h^1 = \text{max}(0, W^1 x + b^1) \\
  h^2 = \text{max}(0, W^2 h^1 + b^2) \\
  y = \text{max}(0, W^3 h^2 + b^3)
  \]
Alternative Graphical Representation

$h^k \rightarrow \max(0, W^{k+1} h^k) \rightarrow h^{k+1}$

$h^k \rightarrow W^{k+1} \rightarrow h^{k+1}$
We want to estimate the parameters, biases and hyper-parameters (e.g., number of layers, number of units) such that we do good predictions.

- Collect a training set of input-output pairs \( \{x, t\} \)
- Encode the output with 1-K encoding \( t = [0, \cdots, 1, \cdots, 0] \)
- Define a loss per training example and minimize the empirical risk

\[
\mathcal{L}(w) = \frac{1}{N} \sum_i \ell(w, x^{(i)}, t^{(i)})
\]

with \( N \) number of examples and \( w \) contains all parameters.
Loss Functions

\[ \mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_i \ell(\mathbf{w}, \mathbf{x}^{(i)}, t^{(i)}) \]

- Probability of class \( k \) given input (softmax):
  \[ p(c_k = 1|\mathbf{x}) = \frac{\exp(y_k)}{\sum_{j=1}^{C} \exp(y_j)} \]

- Cross entropy is the most used loss function for classification
  \[ \ell(\mathbf{x}, t, \mathbf{w}) = - \sum_i t^{(i)} \log p(c_i|\mathbf{x}) \]

- Use gradient descent to train the network
  \[ \min_{\mathbf{w}} \frac{1}{N} \sum_i \ell(\mathbf{w}, \mathbf{x}^{(i)}, t^{(i)}) \]
Backpropagation

- Efficient computation of the gradients by applying the chain rule

\[
x \rightarrow \max(0, W_1^T x + b^1) \rightarrow \max(0, W_2^T h^1 + b^2) \rightarrow W_3^T h^2 + b^3 \rightarrow y
\]

\[
p(c_k = 1|x) = \frac{\exp(y_k)}{\sum_{j=1}^{C} \exp(y_j)}
\]

\[
\ell(x, t, w) = -\sum_i t^{(i)} \log p(c_i|x)
\]

- Compute the derivative of loss w.r.t. the output

\[
\frac{\partial \ell}{\partial y} = p(c|x) - t
\]

- Note that the forward pass is necessary to compute \( \frac{\partial \ell}{\partial y} \)
Backpropagation

- Efficient computation of the gradients by applying the chain rule

\[
\begin{align*}
  x & \rightarrow \max(0, W_1^T x + b^1) \rightarrow \max(0, W_2^T h^1 + b^2) \rightarrow W_3^T h^2 + b^3 \\
  \frac{\partial \ell}{\partial h^2} & = \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial h^2} = (W_3^T)^T (p(c|x) - t) \\
  \frac{\partial \ell}{\partial W_3} & = \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial W_3} = (p(c|x) - t) (h^2)^T \\
  \frac{\partial \ell}{\partial y} & = p(c|x) - t
\end{align*}
\]

- We have computed the derivative of loss w.r.t the output

\[
\frac{\partial \ell}{\partial y} = p(c|x) - t
\]

- Given \( \frac{\partial \ell}{\partial y} \) if we can compute the Jacobian of each module

\[
\begin{align*}
  \frac{\partial \ell}{\partial W_3} &= \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial W_3} = (p(c|x) - t) (h^2)^T \\
  \frac{\partial \ell}{\partial h^2} &= \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial h^2} = (W_3^T)^T (p(c|x) - t)
\end{align*}
\]

- Need to compute gradient w.r.t. inputs and parameters in each layer
Backpropagation

- Efficient computation of the gradients by applying the chain rule

\[
\begin{align*}
\frac{\partial l}{\partial h^2} &= \frac{\partial l}{\partial y} \frac{\partial y}{\partial h^2} = (W^3)^T (p(c|x) - t) \\
\frac{\partial l}{\partial W^2} &= \frac{\partial l}{\partial h^2} \frac{\partial h^2}{\partial W^2} \\
\frac{\partial l}{\partial h^1} &= \frac{\partial l}{\partial h^2} \frac{\partial h^2}{\partial h^1}
\end{align*}
\]

- Given \( \frac{\partial l}{\partial h^2} \) if we can compute the Jacobian of each module
Toy Code (Matlab): Neural Net Trainer

```matlab
% F-PROP
for i = 1 : nr_layers - 1
    [h{i} jac{i}] = nonlinearity(W{i} * h{i-1} + b{i});
end
h{nr_layers-1} = W{nr_layers-1} * h{nr_layers-2} + b{nr_layers-1};
prediction = softmax(h{l-1});

% CROSS ENTROPY LOSS
loss = - sum(sum(log(prediction) .* target)) / batch_size;

% B-PROP
dh{l-1} = prediction - target;
for i = nr_layers - 1 : -1 : 1
    Wgrad{i} = dh{i} * h{i-1}';
    bgrad{i} = sum(dh{i}, 2);
    dh{i-1} = (W{i}' * dh{i}) .* jac{i-1};
end

% UPDATE
for i = 1 : nr_layers - 1
    W{i} = W{i} - (lr / batch_size) * Wgrad{i};
    b{i} = b{i} - (lr / batch_size) * bgrad{i};
end
```