Neural Networks and Deep Learning

Dr. Fayyaz Minhas

Department of Computer Science
University of Warwick
https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs909/
Lecture Plan

• What are Neural Networks?
  – Fundamentals
    • Connectionism
    • Multilayer Perceptron
    • Understanding issues in MLP
    • Making neural networks in Keras
• Why go Deep?
• Issues in deep learning
  – How to solve those issues?
• Modern practices in Deep Learning
  – Convolutional Neural Networks
  – Residual Networks
  – Generative Models
    • Auto-encoders: VAE, NAE
    • Generative Adversarial Networks
    • Recurrent Neural Networks
  – Recurrent Models: RNN, LSTM
  – Neural Networks with Stochastic Depth
  – Other architectures: Ladder, Highway, etc.
  – Transfer Learning
  – Zero-shot and One-shot learning
  – Non-Neural Deep Learning
    • Multilayer Kernel Machines
    • Convolutional Kernel Networks: https://arxiv.org/abs/1406.3332
    • When Correlation Filters Meet Convolutional Neural Networks for Visual Tracking
• Applications
  – Biomedical applications of deep learning
• BASIC NOTES: https://ml-cheatsheet.readthedocs.io/en/latest/nn_concepts.html
Biological Neurons and Networks
Single Neuron

- An abstraction of the biological neuron

\[ y_i = f(u_i) = \sum_j w_{ij} x_j \]
Activation Functions

- Can use any activation function

**Activation Functions**

- **Sigmoid**
  \[ \sigma(x) = \frac{1}{1+e^{-x}} \]

- **tanh**
  \[ \tanh(x) \]

- **ReLU**
  \[ \max(0, x) \]

- **Leaky ReLU**
  \[ \max(0.1x, x) \]

- **Maxout**
  \[ \max(w_1^T x + b_1, w_2^T x + b_2) \]

- **ELU**
  \[
  \begin{cases} 
  x & x \geq 0 \\
  \alpha(e^x - 1) & x < 0 
  \end{cases}
  \]
Neural Networks

• Evaluation
  – Error between predicted and target output
    • Predicted output: $y = f(u) = f(w^T x)$
    • Target output: $t$
    • Error: $(t - y)^2$

• Optimization
  – Whenever the weights change, the output will change
  – Optimize the weights so that the output matches the target
  – Gradient Descent
PyTorch Code

- https://github.com/foxtrotmike/CS909/blob/master/barebones.ipynb

- Review and understand
Single to Multiple Neurons
Multilayer Perceptron: Representation

- Consists of multiple layers of neurons
- Layers of units other than the input and output are called hidden units
- Unidirectional weight connections and biases
- Activation functions
  - Use of activation functions
    - Sigmoidal activations
    - Nonlinear Operation: Ability to solve practical problems
    - Differentiable: Makes theoretical assessment easier
    - Derivative can be expressed in terms of functions themselves: Computational Efficiency
  - Activation function is the same for all neurons in the same layer
  - Input layer just passes on the signal without processing (linear operation)

\[
\begin{align*}
    z_j &= f(z_{in_j}) \\
    z_{in_j} &= \sum_{i=0}^{n} x_i v_{ij}, \quad x_0 = 1, j = 1 \ldots p \\
    y_k &= f(y_{in_k}) \\
    y_{in_k} &= \sum_{j=0}^{p} z_j w_{jk}, \quad z_0 = 1, k = 1 \ldots m
\end{align*}
\]
Why connect neurons?

- Single neuron is really limited
  - Only linear classification

- Multiple layer of Neurons
  - Can get a richer representation

- Can be MIMO
  - Multiple Input, Multiple Output

- Nonlinearity
  - Adding non-linearity can allow us to perform classification of linearly non-separable data
Multilayer Perceptron: Evaluation

- Compute the error between prediction and target
  - SSE Loss: \( \text{loss} = \sum_i \sum_{k=1}^{m} (y_k^i - t_k^i)^2 \)

<table>
<thead>
<tr>
<th>symbol</th>
<th>name</th>
<th>equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_1 )</td>
<td>L(_1) loss</td>
<td>( |y - o|_1 )</td>
</tr>
<tr>
<td>( L_2 )</td>
<td>L(_2) loss</td>
<td>( |y - o|_2^2 )</td>
</tr>
<tr>
<td>( L_1 \circ \sigma )</td>
<td>expectation loss</td>
<td>( |y - \sigma(o)|_1 )</td>
</tr>
<tr>
<td>( L_2 \circ \sigma )</td>
<td>regularised expectation loss(^1)</td>
<td>( |y - \sigma(o)|_2^2 )</td>
</tr>
<tr>
<td>( L_\infty \circ \sigma )</td>
<td>Chebyshev loss</td>
<td>( \max_j</td>
</tr>
<tr>
<td>hinge</td>
<td>hinge [13] (margin) loss</td>
<td>( \sum_j \max(0, \frac{1}{2} - y^{(j)}o^{(j)}) )</td>
</tr>
<tr>
<td>hinge(^2)</td>
<td>squared hinge (margin) loss</td>
<td>( \sum_j \max(0, \frac{1}{2} - y^{(j)}o^{(j)})^2 )</td>
</tr>
<tr>
<td>hinge(^3)</td>
<td>cubed hinge (margin) loss</td>
<td>( \sum_j \max(0, \frac{1}{2} - y^{(j)}o^{(j)})^3 )</td>
</tr>
<tr>
<td>log</td>
<td>log (cross entropy) loss</td>
<td>(- \sum_j y^{(j)} \log \sigma(o)^{(j)} )</td>
</tr>
<tr>
<td>log(^2)</td>
<td>squared log loss</td>
<td>(- \sum_j [y^{(j)} \log \sigma(o)^{(j)}]^2 )</td>
</tr>
<tr>
<td>tan</td>
<td>Tanimoto loss</td>
<td>( -\sum_j \sigma(o)^{(j)}y^{(j)} - \frac{|\sigma(o)|_2^2 + |y|_2^2 - \sum_j \sigma(o)^{(j)}y^{(j)} |\sigma(o)|_2 |y|_2}{\sum_j \sigma(o)^{(j)}y^{(j)} |\sigma(o)|_2 |y|_2} )</td>
</tr>
<tr>
<td>( D_{\text{CS}} )</td>
<td>Cauchy-Schwarz Divergence [3]</td>
<td>(- \log \frac{\sum_j \sigma(o)^{(j)}y^{(j)} |\sigma(o)|_2 |y|_2}{\sum_j \sigma(o)^{(j)}y^{(j)} |\sigma(o)|_2 |y|_2} )</td>
</tr>
</tbody>
</table>
Multilayer Perceptron: Optimization

• Non-convex optimization
  – Because:
    \[ y_{in_k} = \sum_{j=0}^{p} z_j w_{jk}, y_k = f(y_{in_k}) \]

• Weighted combination of activation function outputs

• Compute the gradient of the error/loss function with respect to each weight of the neural network

• Update weights using gradient descent or other methods

   \[ w_{jk}^{new} \leftarrow w_{jk}^{old} - \alpha \frac{\partial l}{\partial w_{jk}^{old}} \quad \text{or} \quad \Delta w_{jk} = -\alpha \frac{\partial l}{\partial w_{jk}^{old}} \]

   \[ v_{ij}^{new} \leftarrow v_{ij}^{old} - \alpha \frac{\partial l}{\partial v_{ij}^{old}} \quad \text{or} \quad \Delta v_{ij} = -\alpha \frac{\partial l}{\partial v_{ij}^{old}} \]
A General Look

• [https://playground.tensorflow.org](https://playground.tensorflow.org)
Multilayer Perceptron

\[ \text{Target} = y - t \]

\[ \text{Loss} \]

\[ \text{Update} \]
Backpropagation training cycle

Feed forward

Weight Update

Backpropagation
Training

• During training we are presented with input patterns and their targets
• At the output layer we can compute the error between the targets and actual output and use it to compute weight updates through the Delta Rule
• But the Error cannot be calculated at the hidden input as their targets are not known
• Therefore we propagate the error at the output units to the hidden units to find the required weight changes (Backpropagation)
• 3 Stages
  – Feed-forward of the input training pattern
  – Calculation and Backpropagation of the associated error
  – Weight Adjustment
• Based on minimization of SSE (Sum of Square Errors)
Proof for the Learning Rule

By the Chain rule, we have:

\[ E = 0.5 \sum_k (t_k - y_k)^2 \]

\[ \frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} 0.5 \sum_k (t_k - y_k)^2 \]

\[ = \frac{\partial}{\partial w_{jk}} 0.5(t_k - y_k)^2 \]

\[ = -(t_k - y_k) \frac{\partial}{\partial w_{jk}} y_k \]

\[ = -(t_k - y_k) \frac{\partial}{\partial w_{jk}} f(y_{in_k}) \]

\[ = -(t_k - y_k)f'(y_{in_k}) \frac{\partial}{\partial w_{jk}} y_{in_k} \]

\[ = -(t_k - y_k)f'(y_{in_k}) \frac{\partial}{\partial w_{jk}} \sum_{j=0}^{p} z_j w_{jk} \]

\[ = -(t_k - y_k)f'(y_{in_k}) z_j = -\delta_k z_j \]

With \( \delta_k = (t_k - y_k)f'(y_{in_k}) \)

\[ \Delta w_{jk} = -\alpha \frac{\partial E}{\partial w_{jk}} = \alpha \delta_k z_j \]

**Use of Gradient Descent Minimization**

\( z_j = f(x_{in_j}), x_{in_j} = \sum_{i=0}^{n} x_i v_{ij}, x_0 = 1, j = 1...p \)

\( y_k = f(0_{in_k}), y_{in_k} = \sum_{j=0}^{p} z_j w_{jk}, x_0 = 1, k = 1...m \)
Proof for the Learning Rule...

\[
\frac{\partial E}{\partial v_{ij}} = \frac{\partial}{\partial v_{ij}} 0.5 \sum_k (t_k - y_k)^2 \\
= 0.5 \sum_k \frac{\partial}{\partial v_{ij}} (t_k - y_k)^2 \\
= \sum_k (t_k - y_k) \frac{\partial}{\partial v_{ij}} (-y_k) \\
= -\sum_k (t_k - y_k) \frac{\partial}{\partial v_{ij}} f(y_{ink}) \\
= -\sum_k (t_k - y_k) f'(y_{ink}) \frac{\partial}{\partial v_{ij}} y_{ink} \\
= -\sum_k \delta_k \frac{\partial}{\partial v_{ij}} \sum_{j=0}^p z_j w_{jk} \\
= -\sum_k \delta_k \frac{\partial}{\partial v_{ij}} z_j w_{jk} = -\sum_k \delta_k w_{jk} \frac{\partial}{\partial v_{ij}} f(z_{inj}) \\
= -\sum_k \delta_k w_{jk} f'(z_{inj}) \frac{\partial}{\partial v_{ij}} z_{inj} \\
= -\sum_k \delta_k w_{jk} f'(z_{inj}) \frac{\partial}{\partial v_{ij}} \sum_{i=0}^n x_i v_{ij} \\
= -\sum_k \delta_k w_{jk} f'(z_{inj}) x_i = -\delta_j x_i \\
\Delta v_{ij} = -\alpha \frac{\partial E}{\partial v_{ij}} = \alpha \delta_j x_i
\]

With \( \delta_k = (t_k - y_k) f'(y_{ink}) \)

Use of Gradient Descent Minimization
Understanding Backpropagation

• Pass the input and compute the output
• Compute Error
• Compute Gradient of error wrt weights
• Compute weight updates
  – Compute \( \delta_k \)
  – “Backpropagate” these \( \delta_k \) through the network to Compute \( \delta_j \)
  – Compute \( \Delta w_{jk} \) and \( \Delta v_{ij} \)
• Update weight updates

\[
\Delta w_{jk} = -\alpha \frac{\partial E}{\partial w_{jk}} = \alpha \delta_k z_j
\]

\[
\delta_k = (t_k - y_k)f'(y_{in_k})
\]

\[
\Delta v_{ij} = -\alpha \frac{\partial E}{\partial v_{ij}} = \alpha \delta_j x_i
\]

\[
\delta_j = \delta_{in_j} f'(z_{in_j})
\]

\[
\delta_{in_j} = \sum_k \delta_k w_{jk}
\]
Step 0. Initialize weights.
    (Set to small random values).
Step 1. While stopping condition is false, do Steps 2–9.
Step 2. For each training pair, do Steps 3–8.

Feedforward:

Step 3. Each input unit \((X_i, i = 1, \ldots, n)\) receives input signal \(x_i\) and broadcasts this signal to all units in the layer above (the hidden units).

Step 4. Each hidden unit \((Z_j, j = 1, \ldots, p)\) sums its weighted input signals,

\[
z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij},
\]

applies its activation function to compute its output signal,

\[
z_j = f(z_{inj}),
\]

and sends this signal to all units in the layer above (output units).

Step 5. Each output unit \((Y_k, k = 1, \ldots, m)\) sums its weighted input signals,

\[
y_{inj} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}
\]

and applies its activation function to compute its output signal,

\[
y_k = f(y_{inj}).
\]
Training Algorithm...

**Backpropagation of error:**

**Step 6.** Each output unit \((Y_k, k = 1, \ldots, m)\) receives a target pattern corresponding to the input training pattern, computes its error information term,

\[
\delta_k = (t_k - y_k)f'(y_{in_k}),
\]

calculates its weight correction term (used to update \(w_{jk}\) later),

\[
\Delta w_{jk} = \alpha \delta_k z_j,
\]

calculates its bias correction term (used to update \(w_{0k}\) later),

\[
\Delta w_{0k} = \alpha \delta_k,
\]

and sends \(\delta_k\) to units in the layer below.
Step 7. Each hidden unit \((Z_j, j=1,\ldots,p)\) sums its delta inputs (from units in the layer above),

\[
\delta_{in_j} = \sum_{k=1}^{m} \delta_k w_{jk},
\]

multiplies by the derivative of its activation function to calculate its error information term,

\[
\delta_j = \delta_{in_j} f'(z_{in_j}),
\]

calculates its weight correction term (used to update \(v_{ij}\) later),

\[
\Delta v_{ij} = \alpha \delta_j x_i,
\]

and calculates its bias correction term (used to update \(v_{0j}\) later),

\[
\Delta v_{0j} = \alpha \delta_j.
\]
Training Algorithm...

Update weights and biases:

Step 8. Each output unit \((Y_k, k = 1, \ldots, m)\) updates its bias and weights \((j = 0, \ldots, p)\):

\[
    w_{jk}(\text{new}) = w_{jk}(\text{old}) + \Delta w_{jk}.
\]

Each hidden unit \((Z_j, j = 1, \ldots, p)\) updates its bias and weights \((i = 0, \ldots, n)\):

\[
    v_{ij}(\text{new}) = v_{ij}(\text{old}) + \Delta v_{ij}.
\]

Step 9. Test stopping condition.

Taken from:
Optimization in minibatches

- We can do a full scale optimization across all examples or take a few examples at a time to determine the gradients
  - Mini-batches
Understanding Backpropagation

• Pass the input and compute the output
• Compute Error
• Compute Gradient of error wrt weights
• Compute weight updates
  – Compute $\delta_k$
  – “Backpropagate” these $\delta_k$ through the network to Compute $\delta_j$
  – Compute $\Delta w_{jk}$ and $\Delta v_{ij}$
• Update weight updates

\[
\Delta w_{jk} = -\alpha \frac{\partial E}{\partial w_{jk}} = \alpha \delta_k z_j
\]

\[
\delta_k = (t_k - y_k)f'(y_{in_k})
\]

\[
\Delta v_{ij} = -\alpha \frac{\partial E}{\partial v_{ij}} = \alpha \delta_j x_i
\]

\[
\delta_j = \delta_{inj}f'(z_{inj})
\]

\[
\delta_{inj} = \sum_k \delta_k w_{jk}
\]
Things to note

• A large number of derivatives will be computed
  – For every input
  – For every weight at every layer

• The update is dependent upon
  • The activation function value
  • The input
  • The target
  • The current weight value
  • The value of the derivative of the activation function of the current layer
  • The value of the derivative of the activation function of the following layers
  • The derivatives are multiplied: Vanishing gradients!
  • The error value
Using Keras

• Simple Exercise using Keras

• https://github.com/foxtrotmike/CS909/blob/master/keras_barebones.ipynb
Using PyTorch

- **Barebones code**
  - [https://github.com/foxtrotmike/CS909/blob/master/barebones.ipynb](https://github.com/foxtrotmike/CS909/blob/master/barebones.ipynb)

- **Using nn-module**
  - [https://github.com/foxtrotmike/CS909/blob/master/pytorch_nn_barebones.ipynb](https://github.com/foxtrotmike/CS909/blob/master/pytorch_nn_barebones.ipynb)
More details on PyTorch

• Using MLP for MNIST classification in PyTorch

  • [https://github.com/foxtrotmike/CS909/blob/master/pytorch_mlp_mnist.ipynb](https://github.com/foxtrotmike/CS909/blob/master/pytorch_mlp_mnist.ipynb)

• Introduces
  – Data Loader
  – Net class
Single to Multiple Neurons
Parameter Selection

• A MLP has a large number of parameters
  – Number of Neurons in Each Layer
  – Number of Layers
  – Activation Function for each neuron: ReLU, logsig...
  – Layer Connectivity: Dense, Dropout...

• Objective function
  – Loss Function: MSE, Entropy, Hinge loss, ...
  – Regularization: L1, L2...

• Optimization Method
  – SGD, ADAM, RMSProp, LM ...
  – Parameters for the Optimization method
    • Weight initialization
    • Momentum, weight decay, etc.
Issues with Neural Networks with non-linear activations

- Unlike an SVM, which has a single global optimum due to its convex loss function, the error surface of a neural network is not as smooth.
- This complicates the optimization.
- A number of “tricks” are used to make the neural network learn.
How to improve MLP?

• Don’t let the network stop learning prematurely!
  – For example: Don’t let the neurons saturate!
  • If the input or the gradient goes to zero, the learning stops!
  – Here is the gradient descent based weight update formula for a 2 layer MLP

\[
\Delta v_{ij} = \alpha x_i f'(v_j^T x) \sum_{k=1}^{m} w_{jk} \left( t_k - f \left( \sum_{j=0}^{p} w_{jk} f(v_j^T x) \right) \right) f' \left( \sum_{j=0}^{p} w_{jk} f(v_j^T x) \right)
\]

\[
\Delta w_{jk} = -\alpha \frac{\partial E}{\partial w_{jk}} = \alpha \delta_k z_j
\]

\[
\delta_k = (t_k - y_k)f'(y_{in_k})
\]

\[
\Delta v_{ij} = -\alpha \frac{\partial E}{\partial v_{ij}} = \alpha \delta_j x_i
\]

\[
\delta_j = \delta_{in_j} f'(z_{in_j})
\]

\[
\delta_{in_j} = \sum_k \delta_k w_{jk}
\]

\[
z_j = f(z_{in_j}), z_{in_j} = \sum_{i=0}^{n} x_i v_{ij}
\]

\[
y_k = f(y_{in_k}), y_{in_k} = \sum_{j=0}^{p} z_j w_{jk}
\]
Zero weight updates

$$\Delta v_{ij} = \alpha x_i f'(v_i^T x) \sum_{k=1}^{m} w_{jk} \left( t_k - f \left( \sum_{j=0}^{p} w_{jk} f(v_j^T x) \right) \right) f' \left( \sum_{j=0}^{p} w_{jk} f(v_j^T x) \right)$$

- When should the weight update be zero?
  - When the neural network’s output matches the target, i.e.,
    $$t_k - f \left( \sum_{j=0}^{p} w_{jk} f(v_j^T x) \right) = 0$$
  - Otherwise, we should be updating weights!

- When can the weight update be zero?
  - $$x_i = 0$$ (don’t use zero input)
  - $$f' = 0$$ (Can happen if the derivative of the activation function becomes zero, e.g., in saturation regions of the sigmoid function.)
  - $$w_{jk} = 0$$ (Don’t start off with zero weights)
  - When input is too large in magnitude (pushes the output into saturation region)
  - When weights are too large (pushes the output into saturation region)
  - Too many layers (products of small derivatives becomes smaller!)
    - Vanishing Gradients
    - Exploding Gradients (when gradient is too large, the weight update will be large and the resulting weight will be large leading to optimization problems)

Glorot & Bengio 2010 “Understanding the difficulty of training deep feedforward neural networks”.
How to improve MLP?

• How to achieve?
  – Weight initialization
    • Use Nguyen-Widrow or more sophisticated weight initialization methods
    • Start with small random weights
    • Large weights will cause saturation
    • Implicit regularization!

<table>
<thead>
<tr>
<th>Binary Xor</th>
<th>RANDOM</th>
<th>NGUYEN-WIDROW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2,891</td>
<td>1,935</td>
</tr>
</tbody>
</table>
How to improve MLP?

• Changes in Data Representation
  – Bipolar inputs/targets are better than binary
    • Zeros in inputs can cause stalls
  – Using clipped bipolar outputs instead of bipolar ones
    • Sigmoidal activation functions will produce a 1.0 or 0.0 only in the asymptote

<table>
<thead>
<tr>
<th></th>
<th>RANDOM</th>
<th>NGUYEN-WIDROW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary XOR</td>
<td>2,891</td>
<td>1,935</td>
</tr>
<tr>
<td>Bipolar XOR</td>
<td>387</td>
<td>224</td>
</tr>
<tr>
<td>Modified bipolar XOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(targets = +0.8 and −0.8)</td>
<td>264</td>
<td>127</td>
</tr>
</tbody>
</table>
How to improve MLP?

• Use slowly saturating or non-saturating nonlinear activation functions
  – Examples: ReLU, Log Activation

Gradients of activation functions

(a) Sigmoid Function and Gradient
(b) Tanh Function and Gradient
(c) ReLU Function and Gradient
(d) Softplus Function and Gradient
How to improve MLP?

- Effect of log activation

<table>
<thead>
<tr>
<th>PROBLEM</th>
<th>LOGARITHMIC</th>
<th>BIPOLAR SIGMOID</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard bipolar XOR</td>
<td>144 epochs</td>
<td>387 epochs</td>
</tr>
<tr>
<td>modified bipolar XOR</td>
<td>77 epochs</td>
<td>264 epochs</td>
</tr>
<tr>
<td>(targets of $+0.8$ or $-0.8$)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Improvements in Optimization

• Use stochastic gradient updates with mini-batches
  — Easy parallelization

• Change learning rate adaptively

• Use momentum \((0 \leq \mu \leq 1)\) based update
  — but too much momentum may cause you to overshoot the local minima

\[
\Delta w_{jk}(t + 1) = \alpha \delta_{kj} z_j + \mu \Delta w_{jk}(t),
\]
Improving MLP

• Data Augmentation
  – Create artificial examples
    • Addition of noise
    • Translation of images or other transforms

• Drop-Off

• Batch Normalization

• Use Early Stopping
  – Keep track of generalization error and stop if the generalization error does not improve enough even when the error on training data is going down
Home/Lab Exercise!

• Solve the XOR using a single hidden layer BPNN with sigmoid activations
  – See what is the effect of different parameters on the convergence characteristics of the neural network
Universal Function Approximation

• A neural network with a single hidden layer is a universal approximator

• Universal Approximation
  – Any function $h(x)$ over $m$ inputs can be represented as follows:
    
    $$
    h(x) = \sum_{i=1}^{P} v_i f(w_i^T x + b_i) + v_0 = \sum_{i=1}^{P} v_i z_i + v_0
    $$

    • $f(\cdot)$ is a non-constant, bounded and monotonically-increasing continuous “basis” function
    • $P$ is the number of functions
    • $h(x)$ is an approximation of $g(x)$, i.e., $|g(x) - h(x)| < \epsilon$
Understanding Universal Approximation

• Let’s do it!

• https://github.com/foxtrotmike/CS909/blob/master/uniapprox.ipynb
Understanding Universal Approximation

• Let’s try to approximate the function $g(x)$ by a NN
• Let’s build a neural network with sigmoid activations in the hidden layer
• The output of a single neuron depends on its net input which is a weighted summation of its inputs (with bias)
• The output is the sum of the outputs of all hidden neurons
• We want to find weights which sum up to produce the target function $g(x) = \sin(4x) + \cos(5x)$
Understanding Universal Approximation

- With a single hidden layer neuron (P=1)
Understanding Universal Approximation

• With a single hidden layer neuron (P=2)
Understanding Universal Approximation

• With a single hidden layer neuron (P=3)
Understanding Universal Approximation

• With a single hidden layer neuron (P=6)
Practical Issues in Universal Approximation

• The universal approximation theorem means that regardless of what function we are trying to learn, we know that a large MLP will be able to represent this function.

• However, we are not guaranteed that the training algorithm will be able to “learn” that function.
  – Optimization can fail
  – Learning is different from optimization
    • The primary requirement for learning is generalization
  – Representability alone does not guarantee learning
Universal Function Approximation

- A neural network with one hidden layer can be used to approximate any shape
  - However, the approximation might require exponentially many neurons
  - How can we reduce the number of computations?

\[ h(x) = \sum_{i=1}^{p} v_i f(w_i^T x + b_i) \]

A single hidden layer NN with step activation is a combination of straight cuts
Total number of learnable parameters: \( pd + p + p \)

The number of required straight cuts to approximate a given shape

How many cuts?

• Remember: Classification can be thought of as partitioning of the feature space

• How can we reduce the number of required cuts?
  – By folding: which is equivalent to:
    • Applying a transformation $\phi(x)$
      – Neural networks
    • Changing the distance metric
      – Distance metric learning
    • Kernelization
      – SVM
Adding more layers is equivalent to transformation

- In the transformed space
- We can implement a learnable feature transformation through neurons!

\[
h(x) = \sum_{i=1}^{p'} v_i f \left( \sum_{j=1}^{d'} w_{ij} g(u_j^T x + c_j) + b_i \right)
\]

Total number of learnable parameters: \(dd' + d' + p'd' + p' + p'\)

Montufar (2014)

Fold and Cut Theorem: [https://www.youtube.com/watch?v=ZREp1mAPKTM](https://www.youtube.com/watch?v=ZREp1mAPKTM)
Width vs. Depth

- An MLP with a single hidden layer is sufficient to represent any function
  - But the layer may be infeasibly large
  - May fail to learn and generalize correctly
- Using a deeper model can reduce the number of units required to represent the desired function and can reduce the amount of generalization error
  - Thus a deeper representation is more efficient!
- A function that could be expressed with $O(n)$ neurons on a network of depth $k$ required at least $O(2^{\sqrt{n}})$ and $O((n - 1)^k)$ neurons on a two-layer neural network: Delalleau and Bengio (2011)
- Functions representable with a deep rectifier net can require an exponential number of hidden units with a shallow (one hidden layer) network: Montufar (2014)
- For a shallow network, the representation power can only grow polynomially with respect to the number of neurons, but for deep architecture, the representation can grow exponentially with respect to the number of neurons: Bianchini and Scarselli (2014)
- Depth of a neural network is exponentially more valuable than the width of a neural network, for a standard MLP with any popular activation functions: Eldan and Shamir (2015)
Width vs. Depth

- Empirical results for some data showed that depth increases generalization performance in a variety of applications.

Figure 6.6: Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses. Data from Goodfellow et al. (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.
Figure 6.7: Deeper models tend to perform better. This is not merely because the model is larger. This experiment from Goodfellow et al. (2014d) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance. The legend indicates the depth of network used to make each curve and whether the curve represents variation in the size of the convolutional or the fully connected layers. We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million. This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together. This could result either in learning a representation that is composed in turn of simpler representations (e.g., corners defined in terms of edges) or in learning a program with sequentially dependent steps (e.g., first locate a set of objects, then segment them from each other, then recognize them).
Comparison of Depth

\[ g(x) = \sin(4x) + \cos(5x) \]

- Both have approximately the same number of parameters (tunable weights)
  - Deeper is better
  - But is difficult to optimize

• 63 neurons in one hidden layer
• 7 neurons in two hidden layers each
REO for MLPs

- **Representation**
  - Defined by the architecture
    - Number of inputs and outputs, Interconnection of neurons, number of neurons in layers, activation functions, etc.
    
    \[
    h(x) = \sum_{i=1}^{p} v_i f(w_i^T x + b_i) + v_0
    \]
    
    - Modern DL libraries require you to define “Representation”
  
- **Evaluation**
  - Defined by the ML problem
  - Can use any loss function
    - Square Error Loss
    - Hinge Loss
    - Cross-Entropy Loss

- **Optimization**
  - Solve for weights that reduce error over training data and (hopefully!) generalize to test data
  - Using any optimization method
    - Stochastic Gradient Descent
    - Adaptive Learning Rate with Momentum (Adam)
    - So many other

Shallow vs. Deep Networks

• Adding more layers increases the representation power of the neural network

• A deep network requires exponentially fewer parameters to get to the same error rate in comparison to a wide neural network
  – More efficient

• However, adding layers leads to a more difficult optimization problem
  – Vanishing and Exploding Gradients

\[
\Delta v_{ij} = \alpha x_i f'(v_j^T x) \sum_{k=1}^{m} w_{jk} \left( t_k - f \left( \sum_{j=0}^{p} w_{jk} f(v_j^T x) \right) \right) f'(\sum_{j=0}^{p} w_{jk} f(v_j^T x))
\]
Where’s Waldo?
Data Mining

Neural Network
Let’s solve it using a neural network

• Input image: 256x256x3
  – Flatten it: 196, 608 dimensional input

• Target: 256x256x3
  – Flatten it: 196, 608 dimensional output

• Let’s use a single hidden layer network
  – Very large number of parameters will be needed

• Let’s use a deep(er) network
  – Still a very large number of parameters will be needed
Convolutional Networks

• A feed-forward network inspired from visual cortex
• Used for image recognition
• Objective
  – Find a set of filters which, when convolved with image, lead to the solution of the desired image recognition task
    • Invariant wrt translation
    • Hierarchical
      – Increasing feature complexity
      – Increasing “Globality”
Basics

• The convolution operation
  – Shows how a function (image) is modified by another (filter)

\[ H(i, j) = \sum_{k=1}^{m} \sum_{l=1}^{n} I(i + k - 1, j + l - 1)K(k, l) \]

https://en.wikipedia.org/wiki/Kernel_(image_processing)
Examples of filters

• Identity Filter

\[
\begin{bmatrix}
0 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\]
Examples of filters

• Edge filters

\[
\begin{bmatrix}
1 & 0 & -1 \\
0 & 0 & 0 \\
-1 & 0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 & 1 & 0 \\
1 & -4 & 1 \\
0 & 1 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
-1 & -1 & -1 \\
-1 & 8 & -1 \\
-1 & -1 & -1
\end{bmatrix}
\]

\[
K = \begin{bmatrix}
9 & -18 & -9 \\
9 & -9 & 0 \\
9 & -9 & 0
\end{bmatrix}
\]

import numpy as np
I = np.array([[1,1,1,1,1],[1,1,10,10,10],[1,1,10,10,10],[1,1,10,10,10],[1,1,10,10,10]])
from scipy.ndimage.filters import convolve
K = np.array([[0,1,0],[1,-4,1],[0,1,0]])
H = convolve(I,K)
### Example Filters

- **Reducing noise using a smoothing filter**

<table>
<thead>
<tr>
<th>Filter Type</th>
<th>Filter Matrix</th>
<th>Example Image</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Box blur</strong></td>
<td>$\frac{1}{9} \begin{bmatrix} 1 &amp; 1 &amp; 1 \ 1 &amp; 1 &amp; 1 \ 1 &amp; 1 &amp; 1 \end{bmatrix}$</td>
<td><img src="image1" alt="Box blur Example" /></td>
</tr>
<tr>
<td>(normalized)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Gaussian blur 3 x 3</strong></td>
<td>$\frac{1}{16} \begin{bmatrix} 1 &amp; 2 &amp; 1 \ 2 &amp; 4 &amp; 2 \ 1 &amp; 2 &amp; 1 \end{bmatrix}$</td>
<td><img src="image2" alt="Gaussian blur 3x3 Example" /></td>
</tr>
<tr>
<td>(approximation)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Gaussian blur 5 x 5</strong></td>
<td>$\frac{1}{256} \begin{bmatrix} 1 &amp; 4 &amp; 6 &amp; 4 &amp; 1 \ 4 &amp; 16 &amp; 24 &amp; 16 &amp; 4 \ 6 &amp; 24 &amp; 36 &amp; 24 &amp; 6 \ 4 &amp; 24 &amp; 16 &amp; 4 &amp; 1 \ 1 &amp; 4 &amp; 6 &amp; 4 &amp; 1 \end{bmatrix}$</td>
<td><img src="image3" alt="Gaussian blur 5x5 Example" /></td>
</tr>
<tr>
<td>(approximation)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(a) Identity kernel

(b) Edge detection kernel

(c) Blur kernel

(d) Sharpen kernel

(e) Lighten kernel

(f) Darken kernel

(g) Random kernel 1

(h) Random kernel 2
Convolution

If you think about it

- Convolution is a sum of products
  - Can be expressed as a dot product

\[
H(i, j) = \sum_{k=1}^{m} \sum_{l=1}^{n} I(i + k - 1, j + l - 1)K(k, l)
\]
How to apply filters?

• The easy way
  – Use skimage filters

```python
import numpy as np
import matplotlib.pyplot as plt

from skimage import filters
from skimage.data import camera

image = camera()
edge_sobel = filters.sobel(image)
plt.figure(); plt.imshow(image, cmap='gray')
plt.figure(); plt.imshow(edge_sobel, cmap='gray')
```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np

class Filter(nn.Module):
    def __init__(self):
        super(Filter, self).__init__()
        self.conv1 = None
    def setKernel(self, K):
        self.conv1 = nn.Conv2d(1,1, K.shape) #initialize a kernel
        # (in_channels, out_channels, kernel_size)
        K = torch.from_numpy(K).float()  # move kernel from numpy to torch by adding extra matrix dimensions
        K = torch.unsqueeze(torch.unsqueeze(K,0),0)  # add extra dimensions for in_channels and out_channels
        self.conv1.weight.data = 0*self.conv1.weight.data + K  # set the kernel as weights (done this way to avoid data type changes)
        self.conv1.bias.data = 0*self.conv1.bias.data  # no bias
        return self
    def forward(self, x):
        x = self.conv1(x)  # perform convolution
        return x

plt.close('all')
from skimage import data
X = data.camera(); plt.imshow(X,cmap='gray',vmin=0,vmax=255) # load and show image
K = np.array([[0 ,1, 0],[1,-4,1], [0, 1 ,0]])  # define filter kernel
X_torch = torch.unsqueeze(torch.unsqueeze(torch.from_numpy(X).float(),0),0)  # move image to torch
f = Filter().setKernel(K)  # set the kernel in Filter object
Z_torch = f(X_torch)  # convolution
Z = Z_torch.squeeze().detach().numpy()  # move back to numpy
plt.figure();plt.imshow(Z,cmap='gray',vmin=0,vmax=255)

https://github.com/foxtrotmike/CS909/blob/master/pytorch_conv.py
Now the interesting question

• Can we learn filters to do something we want to do?
  – Let’s say we have an image and it’s output after a certain operation
  – Can we learn a filter that produces the output given the input?
Example

• Let’s say, we have an image and we want to design a filter that when convolved with the image leads to the desired output. How?
How can this be done?

• Let’s try to build a multi-layer perceptron
  – Input image size: (122,200)
    • This means the number of input neurons will be 24400
  – Target image size: (122,200)
    • This means the number of output neurons will be 24400
  – Number of weights:
    • 24400*24400 = 595,360,000
  – Add hidden layers!
  – Good luck!
Let’s try to learn a 5x5 filter

- **Representation**

\[
H = I \ast K \\
H(i, j) = \sum_{k=1}^{m=5} \sum_{l=1}^{n=5} I(i + k - 1, j + l - 1)K(k, l)
\]

- **Evaluation**

\[
E(K) = \frac{1}{MN} \sum_{k=1}^{M} \sum_{l=1}^{N} (H(i, j) - T(i, j))^2
\]

- **Optimization**

- Solve the following problem: \( \min_K E(K) \)
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np
import skimage

class Filter(nn.Module):
    def __init__(self):
        super(Filter, self).__init__()
        self.conv1 = nn.Conv2d(1, 1, 5)

    def forward(self, x):
        x = self.conv1(x)
        return x

X = skimage.io.imread('in.jpg')/255
T = np.zeros(X.shape, dtype=np.float) #create the target by putting 1.0 at target object locations
T[21,121]=1.0
T[34,36]=1.0
T[64,78]=1.0
T[83,142]=1.0

T = T[2:-2,2:-2] # reduce target filter size to compensate for padding loss in convolution
f = Filter()
optimizer = torch.optim.SGD(f.parameters(), lr=1e-1)
T_torch = torch.from_numpy(T).float()
X_torch = torch.unsqueeze(torch.unsqueeze(torch.from_numpy(X).float(),0),0)

L = []
for _ in range(500):
    Z_torch = f(X_torch).squeeze()
    Z_torch = torch.sigmoid(Z_torch) #output, we have used a sigmoid at the output

    loss = torch.mean((T_torch-Z_torch)**2) #error
    optimizer.zero_grad() #optimization
    loss.backward()
    optimizer.step()
    L.append(loss.item())
Some postprocessing is needed

```python
output = Z_torch.squeeze().detach().numpy()
output = output**2 # contrast stretching
output = (output-np.min(output))/(np.max(output)-np.min(output)) # rescale

plt.figure();plt.imshow(X,cmap='gray');plt.title('input');plt.colorbar()
plt.figure();plt.imshow(T,cmap='gray');plt.title('target');plt.colorbar()
plt.figure();plt.imshow(output,cmap='gray');plt.title('output');plt.colorbar()
plt.figure();plt.imshow(output>0.8,cmap='gray');plt.title('thresholded output')
plt.figure();plt.plot(np.log10(L));plt.title('loss function')
plt.figure();plt.imshow(f.conv1.weight.squeeze().detach().numpy(),cmap='gray');plt.title('learned filter');plt.colorbar()
```

https://github.com/foxtrotmike/CS909/blob/master/learn_filters.py
The learned filter
Another way of looking at this

- We learned a convolution filter kernel based on an input and a target image.
- The filter will act as a + detector when convolved with a new image (hopefully!)
Most basic convolutional neural network

• Acts as a “detection” or “feature extraction” unit
Classification with Multilayer Perceptron

\[ \text{Target} = y - t \]

\[ \text{Loss} \]
Convolutional Neural Networks for ML

• If we want to use the output of convolution filters for learning to classify or regress or rank or for any other task
  – We can use a multilayer perceptron but:
    • We will need to “flatten” the output of the correlation filter (aka feature/filter map)
      – Convert an image to a vector e.g., (8x8 to 64)
    • We will also need to reduce the dimensions of the output
      – Done through “Pooling”
        » Average or max
      – And/Or “Striding”
        » How we move the convolution filter
Most basic convolutional neural network for ML

Detector

Classifier

Input

Filter

Feature Map

Pooled

Flattened

Output

Target

$y - t$

Update

Update

Loss
Filters as automatic feature detectors

• Instead of training a single filter
• We can train a bank of filters each of which acts as a feature detector
• Use a deeper MLP too!
• Convolution Neural Networks!!!
Structure

• Increasing “globality”
  – Input → Convolution → Non-linearity → Sub-sampling ... → Fully Connected Layer (for classification)
# Convolutional neural network (two convolutional layers)

class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.fc = nn.Linear(7*7*32, num_classes)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc(out)
        return out

https://github.com/yunjey/pytorch-tutorial
model = ConvNet(num_classes).to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Train the model
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, i+1, total_step, loss.item()))

# Test the model
model.eval()  # eval mode
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {}%'.format(100 * correct / total))

# Save the model checkpoint
torch.save(model.state_dict(), 'model.ckpt')
Why do CNNs work?

• Local weight connectivity
  – In contrast to a fully connected neural network like a multilayer perceptron, a filter in a CNN operates over an image at the local level

• Shared weights
  – No separate weights for each pixel/block

• Hierarchical representations
Only connect to 9 input, not fully connected

Less parameters!

6 x 6 image

Filter 1

1  -1  -1
-1   1  -1
-1  -1   1

1  0  0  0  0  0
0  1  0  0  1  0
0  0  1  1  0  0
1  0  0  0  1  0
0  1  0  0  1  0
0  0  1  0  1  0

Data Mining

University of Warwick
Filter 1:

\[
\begin{array}{cccccc}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & \\
0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

6 x 6 image

Less parameters!

Even less parameters!

Shared weights
Generic components ("layers"), less domain knowledge

Repeat elementary layers => Going deeper
Deep Learning: Learning Hierarchical Representations

It's deep if it has more than one stage of non-linear feature transformation

Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]
[ConvNetJS demo: training on CIFAR-10]

ConvNetJS CIFAR-10 demo

Description

This demo trains a Convolutional Neural Network on the CIFAR-10 dataset in your browser, with nothing but Javascript. The state of the art on this dataset is about 90% accuracy and human performance is about 94% (not perfect as the dataset can be a bit ambiguous). I used this python script to parse the original files (python version) into batches of images that can be easily loaded into page DOM with img tags.

This dataset is more difficult and it takes longer to train a network. Data augmentation includes random flipping and random image shifts by up to 2px horizontally and vertically.

By default, in this demo we're using Adadelta which is one of per-parameter adaptive step size methods, so we don't have to worry about changing learning rates or momentum over time. However, I still included the text fields for changing these if you'd like to play around with SGD+Momentum trainer.

Report questions/bugs/suggestions to @karpathy.

http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html
Padding

Input

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>9</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>8</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Dimension: 6 x 6

Filter

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

Parameters:
- Size: \( f = 3 \)
- Stride: \( s = 2 \)
- Padding: \( p = 1 \)

Result

\[ = 0 \times 1 + 0 \times 0 + 0 \times (-1) + 0 \times 1 + 4 \times 0 + 9 \times (-1) + 0 \times 1 + 9 \times 0 + 8 \times (-1) = -15 \]

Multiple Channels

Input

\( \begin{bmatrix} 4 & 9 & 2 & 5 & 8 & 3 \\ 5 & 6 & 2 & 4 & 0 & 3 \\ 2 & 4 & 5 & 4 & 5 & 2 \\ 5 & 6 & 5 & 4 & 7 & 8 \\ 5 & 7 & 7 & 9 & 2 & 1 \\ 5 & 8 & 5 & 3 & 8 & 4 \end{bmatrix} \)

Filter

\( \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 1 & -1 \end{bmatrix} \)

Parameters:
- Size: \( f = 3 \)
- #channels: \( n_c = 3 \)
- Stride: \( s = 3 \)
- Padding: \( p = 0 \)

\( \begin{bmatrix} 4 & 9 & 2 & 5 & 8 & 3 \\ 5 & 6 & 2 & 4 & 0 & 3 \\ 2 & 4 & 5 & 4 & 5 & 2 \\ 5 & 6 & 5 & 4 & 7 & 8 \\ 5 & 7 & 7 & 9 & 2 & 1 \\ 5 & 8 & 5 & 3 & 8 & 4 \end{bmatrix} \)

Output

\( \begin{bmatrix} 9 & 2 & 5 & 8 & 3 \\ 6 & 2 & 4 & 0 & 3 \\ 4 & 5 & 4 & 5 & 2 \\ 5 & 6 & 5 & 4 & 7 & 8 \\ 5 & 7 & 7 & 9 & 2 & 1 \\ 5 & 8 & 5 & 3 & 8 & 4 \end{bmatrix} \)

Multiple IO Channels

Input

\( \begin{bmatrix} 4 & 9 & 2 & 5 & 8 & 3 \\ 5 & 6 & 2 & 4 & 0 & 3 \\ 2 & 4 & 5 & 4 & 5 & 2 \\ 5 & 6 & 5 & 4 & 7 & 8 \\ 5 & 7 & 7 & 9 & 2 & 1 \\ 5 & 8 & 5 & 3 & 8 & 4 \end{bmatrix} \)

Filter

\( \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 1 & -1 \end{bmatrix} \)

Parameters:
- Size: \( f = 3 \)
- #channels: \( n_c = 3 \)
- Stride: \( s = 2 \)
- Padding: \( p = 0 \)

\( \begin{bmatrix} 4 & 9 & 2 & 5 & 8 & 3 \\ 5 & 6 & 2 & 4 & 0 & 3 \\ 2 & 4 & 5 & 4 & 5 & 2 \\ 5 & 6 & 5 & 4 & 7 & 8 \\ 5 & 7 & 7 & 9 & 2 & 1 \\ 5 & 8 & 5 & 3 & 8 & 4 \end{bmatrix} \)

1x1 Convolution

Input

\( \begin{bmatrix} 4 & 9 & 2 & 5 & 8 & 3 \\ 5 & 6 & 2 & 4 & 0 & 3 \\ 5 & 7 & 7 & 9 & 2 & 1 \\ 5 & 8 & 5 & 3 & 8 & 4 \end{bmatrix} \)

Filter

\( \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \)

Parameters:
- Size: \( f = 1 \)
- #channels: \( n_c = 1 \)
- Stride: \( s = 1 \)

\( \begin{bmatrix} 4 & 9 & 2 & 5 & 8 & 3 \\ 5 & 6 & 2 & 4 & 0 & 3 \\ 5 & 7 & 7 & 9 & 2 & 1 \\ 5 & 8 & 5 & 3 & 8 & 4 \end{bmatrix} \)
Important Concepts

• Differences from fully connected nets
  – 3D volume of neurons
  – Local connectivity
  – Shared weights

• Hyper-parameter
  – Number of filters
  – Filter shape (receptive field)
  – Pooling type and shape

• Regularization
  • Dropout
  • Data Augmentation
  • Early Stopping
  • Norm constraints
  • L1/L2 regularization

  – Use performance over a validation set to pick hyperparameters
Important Concepts

• Optimization

For each of the N epochs

Pick a batch of n examples at one time (Data Loader)
Pass them through the network and compute outputs
Compute Loss
Backpropagate (Compute Gradients, Update Weights)

Compute error over a validation data set for “early stopping”

• Early stopping: Pick the weights of the neural network where validation error is the lowest
• Always, plot the loss over training and validation data across epochs – Give you an idea of whether the neural network is learning or not
Famous CNN

• LeNet (Le Cunn 1990, 1998)

• AlexNet
Famous CNN

- VGG19
- Inception
- XCeption
Transfer Learning

• Use a pretrained network for one task
• Keep the convolutional layers fixed
• Train the last layers (classification) for your task
Predicting Hurricane Intensities

• Deep-PHURIE

PHURIE vs Deep-PHURIE Comparisons

![Bar chart comparing PHURIE and Deep-PHURIE across years 2001 to 2015, with the mean values also shown.](chart.png)
Deep-PHURIE Robustness Analysis
Activation Maps for Deep PHURIE
Regularization Mechanisms

- L2 penalty to weights
  - Weight_decay parameter

- Handling vanishing (or exploding) gradients
  - Pre-training (old!)
  - Drop-out
  - Batch Normalization

```python
nn.Dropout(0.5)
nn.BatchNorm2d(6)
```
Understanding Drop-out

“Dropout: A Simple Way to Prevent Neural Networks from Overfitting” by Srivastava et al., 2014.

- Randomly drop units (along with their connections) from the neural network during training
- Average weights across all “thinned” networks
- Replaces explicit regularization and produces faster learning
# Effect of Dropout

## 6.1.1 MNIST

<table>
<thead>
<tr>
<th>Method</th>
<th>Unit Type</th>
<th>Architecture</th>
<th>Error %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Neural Net (Simard et al., 2003)</td>
<td>Logistic</td>
<td>2 layers, 800 units</td>
<td>1.60</td>
</tr>
<tr>
<td>SVM Gaussian kernel</td>
<td>NA</td>
<td>NA</td>
<td>1.40</td>
</tr>
<tr>
<td>Dropout NN</td>
<td>Logistic</td>
<td>3 layers, 1024 units</td>
<td>1.35</td>
</tr>
<tr>
<td>Dropout NN</td>
<td>ReLU</td>
<td>3 layers, 1024 units</td>
<td>1.25</td>
</tr>
<tr>
<td>Dropout NN + max-norm constraint</td>
<td>ReLU</td>
<td>3 layers, 1024 units</td>
<td>1.06</td>
</tr>
<tr>
<td>Dropout NN + max-norm constraint</td>
<td>ReLU</td>
<td>3 layers, 2048 units</td>
<td>1.04</td>
</tr>
<tr>
<td>Dropout NN + max-norm constraint</td>
<td>ReLU</td>
<td>2 layers, 4096 units</td>
<td>1.01</td>
</tr>
<tr>
<td>Dropout NN + max-norm constraint</td>
<td>ReLU</td>
<td>2 layers, 8192 units</td>
<td>0.95</td>
</tr>
<tr>
<td>Dropout NN + max-norm constraint (Goodfellow et al., 2013)</td>
<td>Maxout</td>
<td>2 layers, (5 × 240) units</td>
<td>0.94</td>
</tr>
<tr>
<td>DBN + finetuning (Hinton and Salakhudtinov, 2006)</td>
<td>Logistic</td>
<td>500-500-2000</td>
<td>1.18</td>
</tr>
<tr>
<td>DBM + finetuning (Salakhudtinov and Hinton, 2009)</td>
<td>Logistic</td>
<td>500-500-2000</td>
<td>0.96</td>
</tr>
<tr>
<td>DBN + dropout finetuning</td>
<td>Logistic</td>
<td>500-500-2000</td>
<td>0.92</td>
</tr>
<tr>
<td>DBM + dropout finetuning</td>
<td>Logistic</td>
<td>500-500-2000</td>
<td>0.79</td>
</tr>
</tbody>
</table>
Understanding Batch-Normalization

• Re-normalization of examples
• Given a batch of N examples, each dimension of each example is normalized to zero mean and unit variance
• Minimizes “covariance shift”
  – a change in the distribution of a function’s domain
  – Input changes and now the function cannot deal with it
  – Layer to layer changes
• More effective than drop-out
• No need for “pre-training”

Input: Values of $x$ over a mini-batch: $B = \{x_1...m\}$; Parameters to be learned: $\gamma, \beta$

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

\[
\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad \text{// mini-batch mean}
\]

\[
\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \quad \text{// mini-batch variance}
\]

\[
\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad \text{// normalize}
\]

\[
y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad \text{// scale and shift}
\]

https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html
Effect of Batch Normalization

Figure taken from [S. Ioffe & C. Szegedy]
Batch Normalization vs Dropout

Reading

• Easy Reading: Machine Learning is Fun! Part 3: Deep Learning and Convolutional Neural Networks

• Required reading
  – ImageNet Classification with Deep Convolutional Neural Networks

• Results of various methods
  – http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html
# Origins of Deep Learning

<table>
<thead>
<tr>
<th>Year</th>
<th>Contributor</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>300 BC</td>
<td>Aristotle</td>
<td>introduced Associationism, started the history of human’s attempt to understand brain.</td>
</tr>
<tr>
<td>1873</td>
<td>Alexander Bain</td>
<td>introduced Neural Groupings as the earliest models of neural network, inspired Hebbian Learning Rule.</td>
</tr>
<tr>
<td>1943</td>
<td>McCulloch &amp; Pitts</td>
<td>introduced MCP Model, which is considered as the ancestor of Artificial Neural Model.</td>
</tr>
<tr>
<td>1949</td>
<td>Donald Hebb</td>
<td>considered as the father of neural networks, introduced Hebbian Learning Rule, which lays the foundation of modern neural network.</td>
</tr>
<tr>
<td>1958</td>
<td>Frank Rosenblatt</td>
<td>introduced the first perceptron, which highly resembles modern perceptron.</td>
</tr>
<tr>
<td>1974</td>
<td>Paul Werbos</td>
<td>introduced Backpropagation</td>
</tr>
<tr>
<td>1980</td>
<td>Teuvo Kohonen</td>
<td>introduced Self Organizing Map</td>
</tr>
<tr>
<td></td>
<td>Kunihiro Fukushima</td>
<td>introduced Neocogitron, which inspired Convolutional Neural Network</td>
</tr>
<tr>
<td>1982</td>
<td>John Hopfield</td>
<td>introduced Hopfield Network</td>
</tr>
<tr>
<td>1985</td>
<td>Hilton &amp; Sejnowski</td>
<td>introduced Boltzmann Machine</td>
</tr>
<tr>
<td>1986</td>
<td>Paul Smolensky</td>
<td>introduced Harmonium, which is later known as Restricted Boltzmann Machine</td>
</tr>
<tr>
<td></td>
<td>Michael I. Jordan</td>
<td>defined and introduced Recurrent Neural Network</td>
</tr>
<tr>
<td>1990</td>
<td>Yann LeCun</td>
<td>introduced LeNet, showed the possibility of deep neural networks in practice</td>
</tr>
<tr>
<td>1997</td>
<td>Schuster &amp; Paliwal</td>
<td>introduced Bidirectional Recurrent Neural Network</td>
</tr>
<tr>
<td></td>
<td>Hochreiter &amp; Schmidhuber</td>
<td>introduced LSTM, solved the problem of vanishing gradient in recurrent neural networks</td>
</tr>
</tbody>
</table>
## Origins of Deep Learning

<table>
<thead>
<tr>
<th>Year</th>
<th>Author(s)</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006</td>
<td>Geoffrey Hinton</td>
<td>introduced Deep Belief Networks, also introduced layer-wise pretraining technique, opened current deep learning era.</td>
</tr>
<tr>
<td>2009</td>
<td>Salakhutdinov &amp; Hinton</td>
<td>introduced Deep Boltzmann Machines</td>
</tr>
<tr>
<td>2012</td>
<td>Geoffrey Hinton</td>
<td>introduced Dropout, an efficient way of training neural networks</td>
</tr>
<tr>
<td>2013</td>
<td>Kingma &amp; Welling</td>
<td>introduced Variational Autoencoder (VAE), which may bridge the field of deep learning and the field of Bayesian probabilistic graphic models.</td>
</tr>
<tr>
<td>2014</td>
<td>Ian J. Goodfellow</td>
<td>introduced Generative Adversarial Network.</td>
</tr>
<tr>
<td>2015</td>
<td>Ioffe &amp; Szegedy</td>
<td>introduced Batch Normalization</td>
</tr>
</tbody>
</table>

Increasing representation power with depth
Issues with Depth

• Generalization
  – Large networks are large capacity machines
  – Remember: Learning requires generalization and goes beyond mere minimization of an objective function!

• Failure to Optimize
  – Random initialization leads to the network being stuck in poor solutions
  – Deeper networks are more prone to vanishing/exploding gradients and optimization failures
    • “Greedy Layer-Wise Training of Deep Networks” by Bengio et al., 2006
      – Uses unsupervised pre-training to initialize the weights of a network such that the optimization becomes easier
    • Since 2010, this has been replaced with Drop-out and batch-normalization schemes which improve the optimization performance
      – Rectified Linear Units get rid of the vanishing gradient problem
      – Drop-out improves generalization
      – Batch Normalization accelerates deep learning and improves generalization

• Large scale optimization is tricky in deep learning
  – Computationally demanding
  – Requires efficient methods
    • Stochastic gradient and sub-gradient methods
Issues with depth

• Handling variety of neural network architectures
  – How can we develop a framework of learning in which we can add layers, have a large diversity of layer connectivity, change objective functions and losses, layer connectivity, regularization, etc.?
  – And still solve the optimization problem in an efficient manner!
  – **Symbolic Computation and Automatic Differentiation**
    • TensorFlow
    • PyTorch
  – **GPU**
    • Efficient matrix operations
    • Higher bandwidth
DL Libraries

• All Deep Learning Libraries Do three things
  – Automatic Differentiation (Efficient Algorithms such as Reverse mode autodiff!)
  – Implement Optimizers
  – Use efficient hardware for multiprocessing (GPUs)
• Support efficient representation / abstraction

• pyTorch
  – Imperative Programming
    • Run, Run, Run...
  – Dynamic Computing Graphs
    • Graph built at run time
    • Build as you go
  – Good for research

• TensorFlow
  – Symbolic
    • Compile then run/fit
  – Static Computing Graphs
    • Build before you go (new version has dynamic graphs too!)
  – Good Documentation
  – Distributed Computing / Delivery
  – TensorFlow.js

• Keras
• CAFFE
• Theano
• JAX
• Julia - FluxML
Deep Learning Libraries

• Essentially Automatic Differentiation Tools with optimization packages
  – Represent a neural network loss calculation as a computational graph and then compute the gradients

• Can use GPU

\[
e = (a + b)(b + 1) = ab + a + b^2 + b
\]

\[
\frac{\partial e}{\partial a}_{(a,b)=(2,1)} = b + 1 = 2
\]

\[
\frac{\partial e}{\partial b}_{(a,b)=(2,1)} = a + 2b + 1 = 5
\]

```python
import torch
import numpy as np
from torchviz import make_dot

a = torch.from_numpy(np.array([2.0])); a.requires_grad_(True)
b = torch.from_numpy(np.array([1.0])); b.requires_grad_(True)
e = (a+b)*(b+1)
e.backward()
print(a.grad) # 2
print(b.grad) # 5
make_dot(e)
```

!pip install torchviz
from torchviz import make_dot
make_dot(tloss,params=dict(model.named_parameters()))
Computation Graph of a two-layer network

```
model = torch.nn.Sequential(
    torch.nn.Linear(2, 2),
    torch.nn.Sigmoid(),
    torch.nn.Linear(2, 1),
    torch.nn.Sigmoid()
).to(device)

z = model(x)

# Manual Gradient Descent
model.zero_grad()
e.backward()

with torch.no_grad():
    for param in model.parameters():
        param.data -= learning_rate * param.grad

# Using Built-in Optimizer
# optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
model.zero_grad()
e.backward()

optimizer.step()
```

```
e = loss_fn(z, y)
```
Optimization Methods

• Sparse data
  – Adaptive Learning rate

• RMSProp, AdaDelta and Adam are very similar
  – Do well in general
  – Adam slightly outperforms RMSProp and is a good choice

• Parallelizing and distributing SGD
  – TensorFlow uses computational graph distribution
  – Other parallel schemes include: HogWild! Delayed SGD, etc.
Optimization Methods

- Gradient Descent: Go down!
  \[ \theta = \theta - \eta \cdot \nabla \theta J(\theta) \]
- Stochastic Gradient Descent
  \[ \theta = \theta - \eta \cdot \nabla \theta J(\theta; x^{(i)}; y^{(i)}) \]
- Mini-batch Gradient Descent
  \[ \theta = \theta - \eta \cdot \nabla \theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \]
- SGD with momentum: accelerate if going downhill for a long time
- Nesterov momentum: accelerate but not indefinitely
- Adagrad: Adaptive Learning Rate by accumulating past gradients
- AdaDelta/RMSProp: Adaptive Learning rate but does not accumulate all past gradients
- Adam: Adaptive learning rate with momentum

An overview of gradient descent optimization algorithms by Sebastian Ruder, 20-16
Types of Neural Networks

• “Fully Connected”/Dense Feed Forward Backpropagation multi-layer perceptrons
• Convolutional neural networks

• Residual Neural networks
• Recurrent neural networks
• Auto-encoders
• Adversarial Networks
• Graph Neural Networks
A mostly complete chart of

Neural Networks

©2016 Fjodor van Veen - asimovinstitute.org
http://www.asimovinstitute.org/neural-network-zoo/

Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool

Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)

Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)

Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Spectrum of Depth

- 5 layers: easy
- >10 layers: initialization, Batch Normalization
- >30 layers: skip connections
- >100 layers: identity skip connections
- >1000 layers: ?

shallower -> deeper
Increasing Depth (10-100 Layers)

• What if we keep on stacking layers?
  – 56-layer net has **higher training error** and test error than 20-layer net

Simply Stacking Layers?

- “Overly deep” plain nets have **higher training error**
- A general phenomenon, observed in many datasets
- Reasons
  - Optimization failure
Residual Learning: skip connections

Plain Network

\[ H(x) = \text{weight layer}(x) \rightarrow \text{relu} \rightarrow \text{weight layer} \]

\[ H(x) \] is any desired mapping
Hope the 2 weight layers fit \( H(x) \)

Residual Network

\[ x \]
\[ \rightarrow \text{weight layer} \rightarrow \text{relu} \rightarrow \text{weight layer} \]

\[ F(x) = H(x) - x \]
\[ H(x) = F(x) + x \]

\[ H(x) \] is any desired mapping
Hope the 2 weight layers fit \( F(x) \)

The network learns fluctuations \( F(x) = H(x) - x \)
Easier!

ResNet Models

• No Dropout
• With Batch Normalization
• Use Data Augmentation
A residual block in code

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, activation='relu'):
        super().__init__()
        self.in_channels, self.out_channels, self.activation = in_channels, out_channels, activation
        self.blocks = nn.Identity()
        self.activate = activation_func(activation)
        self.shortcut = nn.Identity()

    def forward(self, x):
        residual = x
        if self.should_apply_shortcut: residual = self.shortcut(x)
        x = self.blocks(x)
        x += residual
        x = self.activate(x)
        return x

    @property
    def should_apply_shortcut(self):
        return self.in_channels != self.out_channels

Strongly recommended
Deep ResNets can be trained without difficulties.

Deeper ResNets have lower training error, and also lower test error.
Deeper ResNets have lower error

this model has lower time complexity than VGG-16/19

ResNet-152: 5.7
ResNet-101: 6.1
ResNet-50: 6.7
ResNet-34: 7.4

10-crop testing, top-5 val error (%)

University of Warwick
ImageNet experiments

ImageNet Classification top-5 error (%)

- ILSVRC'15 ResNet: 3.57
- ILSVRC'14 GoogleNet: 6.7
- ILSVRC'14 VGG: 7.3
- ILSVRC'13: 11.7
- ILSVRC'12 AlexNet: 16.4
- ILSVRC'11: 25.8
- ILSVRC'10: 28.2

152 layers
ResNet Results

• **1st places in all five main tracks**
  • ImageNet Classification: “Ultra-deep” 152-layer nets
  • ImageNet Detection: 16% better than 2nd
  • ImageNet Localization: 27% better than 2nd
  • COCO Detection: 11% better than 2nd
  • COCO Segmentation: 12% better than 2nd
Residual Networks

• **Required Reading**
  

• **Many third-party implementations**
  
  - list in [https://github.com/KaimingHe/deep-residual-networks](https://github.com/KaimingHe/deep-residual-networks)
  - Torch ResNet: [https://github.com/pytorch/examples/tree/master/imagenet](https://github.com/pytorch/examples/tree/master/imagenet)
  - Transfer Learning with ResNet: [https://www.pluralsight.com/guides/introduction-to-resnet](https://www.pluralsight.com/guides/introduction-to-resnet)

Supervised vs Unsupervised Learning

**Supervised Learning**

- **Data**: \((x, y)\)
- \(x\) is data, \(y\) is label
- **Goal**: Learn a *function* to map \(x \rightarrow y\)
- *Classification*  
- *Regression*  

**Unsupervised Learning**

- **Data**: \(x\)
- Just data, no labels!
- **Goal**: Learn some underlying hidden *structure* of the data
- *Clustering*
- *Dimensionality Reduction*
Unsupervised Learning - Autoencoders

L2 Loss function:
\[ \| x - \hat{x} \|^2 \]

Reconstructed input data

Encoder: 4-layer conv
Decoder: 4-layer upconv

Reconstructed data

Input data

Features

Decoder

Input data
Autoencoder

Unsupervised approach for learning a lower-dimensional feature representation from unlabelled training data

28 x 28 = 784

Usually <784

Compact representation of the input object

reconstruct the original object

Q: Why dimensionality reduction?

A: Want features to capture meaningful factors of variation in data
How to Train Autoencoders?

Train such that features can be used to reconstruct original data “Autoencoding” – encoding itself

Minimize \((x - \hat{x})^2\)

As close as possible

Output of the hidden layer is the code

Bottleneck later

Output of the hidden layer is the code
Deep Auto-encoder

• Of course, the auto-encoder can be deep

As close as possible

Initialise by Restricted Boltzmann Machine (RMB) layer-by-layer

Symmetric is not necessary.

A generative look at Machine Learning

https://www.youtube.com/watch?v=Ow25mjFjSmg.
Generating data with machine learning

• Can we generate examples?
• How?
  – Density Modelling
    • Modelling the Probability of observing a given point \( p(x) \)
    • Once I have an explicit or implicit \( p(x) \), I can sample from that distribution to generate an example
    • How to learn \( p(x) \)?
      – We can use a neural network!
        – \( p_{model}(x; \theta) \)
  – Relation to partitioning of the feature space
Generative Models

• Can we build a model to approximate a data distribution?

Want to learn $p_{\text{model}}(x)$ similar to $p_{\text{data}}(x)$

Density estimation, a core problem in unsupervised learning

Several flavors:
- Explicit density estimation: explicitly define and solve for $p_{\text{model}}(x)$
- Implicit density estimation: learn model that can sample from $p_{\text{model}}(x)$ w/o explicitly defining it

https://openai.com/blog/generative-models/
Generating Data with Autoencoders

As close as possible

Input Layer

Layer

Layer

\( W_1 \)

Layer

Layer

\( W_2 \)

Layer

Layer

Layer

Layer

Output Layer

Code

\( x \)

\( \hat{x} \)

Input Layer

Layer

Layer

\( W_1 \)

Layer

Layer

Layer

Layer

Output Layer

Code

\( x \)
Generative Models

• Can generate data

---

A Simple Generative Machine Learning Example

• Nature
  – A coin with \( p(x=H)=0.7 \) and \( p(x=T)=0.3 \)
  – Generates data

• Data
  – \{H,H,H,T,T,H,T,H,H,T\}

• Goal of Generative Learning
  – Make a machine learning model that can generate data (heads or tails) that follows the same distribution as the real world or natural process
  – Assume you have a learning machine \( G \), which generates data then difference between the probability distributions of real and generated samples should be small
Density Modelling in Generative Models

– Explicit Density Modelling
  • The model should be able to
    – Generate a sample $x$
    – Tell what is the probability of $p(x=H)$
  • Examples
    – Variational Autoencoders

– Implicit Density Modelling
  • The model should be able to
    – Generate a sample $x$
  • Examples
    – Generative Adversarial Networks
REO for Generative Models

• Goal
  – Learn parameters of the model so that the generated examples follow the same distribution as the real world examples

• Representation: \( x = f(z; \theta) \)

• Evaluation: Differences between the probability distribution of \( x \) in nature, \( p(x) \) and of the generated samples \( p_\theta(x) \) from \( f(z; \theta) \) should be minimized
  – So that if I sample from \( p(x) \) or if I sample from \( p_\theta(x) \), the real and generated samples are similar

• Optimization
Generative Adversarial Networks

• A generator learns to generate data that “fits” the unknown underlying probability distribution of a training set based on whether a “discriminator” can distinguish between the samples from the wild and the generated ones.
GANs Applications

• GANs have some impressive applications
  – Synthetic Image Generation
  – Speech Generation
  – Image to Image Translation
  – Style Transfer
  – Deep Fakes

Barebones GAN
https://github.com/foxtrotmike/CS909/blob/master/simpleGAN.ipynb


https://github.com/eriklindernoren/PyTorch-GAN
https://affinelayer.com/pixsrv/
The GAN Zoo

- **GAN** - Generative Adversarial Networks
- **3D-GAN** - Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling
- **acGAN** - Face Aging With Conditional Generative Adversarial Networks
- **AC-GAN** - Conditional Image Synthesis With Auxiliary Classifier GANs
- **AdaGAN** - AdaGAN: Boosting Generative Models
- **AEGAN** - Learning Inverse Mapping by Autoencoder based Generative Adversarial Nets
- **AttGAN** - Amortised Map Inference for Image Super-resolution
- **AL-CGAN** - Learning to Generate Images of Outdoor Scenes from Attributes and Semantic Layouts
- **ALI** - Adversarially Learned Inference
- **AM-GAN** - Generative Adversarial Nets with Labeled Data by Activation Maximization
- **AnoGAN** - Unsupervised Anomaly Detection with Generative Adversarial Networks to Guide Marker Discovery
- **ArtGAN** - ArtGAN: Artwork Synthesis with Conditional Categorical GANs
- **b-GAN** - b-GAN: Unified Framework of Generative Adversarial Networks
- **Bayesian GAN** - Deep and Hierarchical Implicit Models
- **BEGAN** - BEGAN: Boundary Equilibrium Generative Adversarial Networks
- **BiGAN** - Adversarial Feature Learning
- **BS-GAN** - Boundary-Seeking Generative Adversarial Networks
- **CGAN** - Conditional Generative Adversarial Nets
- **CaloGAN** - CaloGAN: Simulating 3D High Energy Particle Showers in Multi-Layer Electromagnetic Calorimeters with Generative Adversarial Networks
- **CCGAN** - Semi-Supervised Learning with Context-Conditional Generative Adversarial Networks
- **CatGAN** - Unsupervised and Semi-supervised Learning with Categorical Generative Adversarial Networks
- **CoGAN** - Coupled Generative Adversarial Networks
- **Context-RNN-GAN** - Contextual RNN-GANs for Abstract Reasoning Diagram Generation
- **C-RNN-GAN** - C-RNN-GAN: Continuous recurrent neural networks with adversarial training
- **CS-GAN** - Improving Neural Machine Translation with Conditional Sequence Generative Adversarial Nets
- **CVAE-GAN** - CVAE-GAN: Fine-Grained Image Generation through Asymmetric Training
- **CycleGAN** - Unpaired Image-to-image Translation using Cycle-Consistent Adversarial Networks
- **DTN** - Unsupervised Cross-Domain Image Generation
- **DCGAN** - Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks
- **DiscoGAN** - Learning to Discover Cross-Domain Relations with Generative Adversarial Networks
- **DR-GAN** - Disentangled Representation Learning GAN for Pose-Invariant Face Recognition
- **DualGAN** - DualGAN: Unsupervised Dual Learning for Image-to-Image Translation
- **ESGAN** - Energy-based Generative Adversarial Network
- **f-GAN** - f-GAN: Training Generative Neural Samplers using Variational Divergence Minimization
- **FF-GAN** - Towards Large-Pose Face Frontalization in the Wild
- **GAIWNN** - Learning What and Where to Draw
- **GeneGAN** - GeneGAN: Learning Object Transfiguration and Attribute Subspace from Unpaired Data
- **Geometric GAN** - Geometric GAN
- **GoGAN** - Gang of GANs: Generative Adversarial Networks with Maximum Margin Ranking
- **GP-GAN** - GP-GAN: Towards Realistic High-Resolution Image Blending
- **IAN** - Neural Photo Editing with IntraSpective Adversarial Networks
- **IGAN** - Generative Visual Manipulation on the Natural Image Manifold
- **lcGAN** - Invertible Conditional GANs for Image editing
- **ID-CGAN** - Image De-rating Using a Conditional Generative Adversarial Network
- **ImprovedGAN** - Improved Techniques for Training GANs
- **InfoGAN** - InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets
- **LAGAN** - Learning Particle Physics by Example: Location-Aware Generative Adversarial Networks for Physics Synthesis
- **LAPGAN** - Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks

[https://github.com/hindupuravinash/the-gan-zoo](https://github.com/hindupuravinash/the-gan-zoo)
Conditional GAN

Training data: \((c, x), \text{ (condition, desired output)}\), e.g., (text, image)

For Generator:
It wants the discriminator to classify \((c, G(c))\) as positive

For Discriminator:
Positive example: \((c, x)\), e.g., the original (text, image) pair

Negative examples: \((c, G(c))\), e.g., (text, generated image) pair \((c', x)\), e.g., (arbitrary text, original image) pair
Text-to-Image Synthesis

### Text to Image – Results

<table>
<thead>
<tr>
<th>Caption</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>a pitcher is about to throw the ball to the batter</td>
<td><img src="" alt="Images" /></td>
</tr>
<tr>
<td>a group of people on skis stand in the snow</td>
<td><img src="" alt="Images" /></td>
</tr>
<tr>
<td>a man in a wet suit riding a surfboard on a wave</td>
<td><img src="" alt="Images" /></td>
</tr>
</tbody>
</table>
Image-to-image Translation

Unpaired Transformation – Cycle GAN, Disco GAN

Transform an object from one domain to another without paired data

Domain X

Domain Y

photo

van Gogh

Monet ↔ Photos

Zebras ↔ Horses

Summer ↔ Winter

Monet → photo

photo → Monet

horse → zebra

summer → winter

winter → summer
How to model language?

- Machine Translation
- Sentiment Classification
- Text Classification
- Chatbots
- Summarization
- Auto-correct
Preprocessing: Tokenization

• Tokenization is a part of pre-process to break a stream of text up into words, phrases, symbols, or other meaningful elements called tokens.

Text
“The cat sat on the mat.”

↓

Tokens
“the”, “cat”, “sat”, “on”, “the”, “mat”, “.”
Preprocessing: Stop word removal

• We usually remove meaningless words from our data called stop words.

• There is a huge list of stopwords in the NLTK library.

• Examples: is, am and are etc.
Pre-processing: lemmatization and stemming

• Reducing tokens to stems or lemmas
  – Stems may have no meaning
  – Lemmas have meaning

• Example:
  ‘survei','network','architectur','emerg','technolog','near','futur','pri me','object','demand','need','address','increas','capac','improv','dat a','rate','decreas','latenc','better', 'qualiti','servic'
Feature Extraction

• Given a set of documents (Corpus)
• Build a dictionary of tokens
• Given a document
  – Represent it by the “term frequency”
    • How many times each term occurs in the document
      – A: 2
      – B: 0
      – C: 4
      – ...

TF-IDF

• Term Frequency Inverse Document Frequency
  – Models the frequency of occurrence in a document relative to all documents

  - \( tf(t, d) = \) frequency of token \( t \) in document \( d \)
  - \( idf(t, D) = \log \frac{\text{Number of documents (D)}}{\text{Number of documents in which } t \text{ appears}} \)
  - \( tfidf(t, d, D) = tf(t, d)idf(t, D) \)

• Leads to sparse features which can then be used for classification
  – Document Classification: Represent each document by the TF-IDF of its terms
Language Models

• Learn a neural representation or embedding of a token by trying to predict what comes ___ a word in a sentence — (after)

• GPT-2
• BERT
Practical Example

• PRISM: Recommending journals for a given paper
“Vanilla” Neural Network

Vanilla Neural Networks

(credit: Fei-Fei Li & Justin Johnson & Serena Yeung)
Recurrent Neural Networks: Process Sequences

Vanilla Neural Networks
e.g. Image Captioning
image -> seq. of words

(e.g. Sentiment Classification
Seq. of words -> sentiment)

(e.g. Machine Translation
seq of words -> seq of words)

(e.g. Video classification on frame level)

(crit: Fei-Fei Li & Justin Johnson & Serena Yeung)
Recurrent Neural Network

We can process a sequence of vectors $\mathbf{x}$ by applying a recurrence formula at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

where $h_t$ is the new state, $h_{t-1}$ is the old state, and $f_W$ is some function with parameters $W$. The input vector at some time step is $x_t$. The diagram illustrates the flow of information through the network.
Recurrent Neural Network

We can process a sequence of vectors $\mathbf{x}$ by applying a recurrence formula at every time step:

$$ h_t = f_W(h_{t-1}, x_t) $$

Notice: the same function and the same set of parameters are used at every time step.
(Simple) Recurrent Neural Network

- The state consists of a single "hidden" vector $h$
- Re-use the same weight matrix at every time-step

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

Sometimes called a "Vanilla RNN" or an "Elman RNN" after Prof. Jeffrey Elman
RNN: Computational Graph: Many to Many

\[ L \]

\[ L_1 \quad L_2 \quad L_3 \quad L_T \]

\[ h_0 \quad h_1 \quad h_2 \quad h_3 \quad \ldots \quad h_T \]

\[ y_1 \quad y_2 \quad y_3 \quad \ldots \quad y_T \]

\[ W_{xh} \quad W_{hh} \quad W_{hy} \quad \ldots \quad W_{xh} \quad W_{hy} \]

\[ x_1 \quad x_2 \quad x_3 \quad \ldots \quad x_T \]
RNN: Computational Graph: Many to One

E.g. sentiment classification

The food is delicious!
RNN: Computational Graph: One to Many

E.g. image captioning

<START> a dog <END>

\[
\begin{align*}
W_{xh} & & & \\
W_{hh} & & & \\
W_{hy} & & & \\
W_{hy} & & & \\
W_{hy} & & & \\
W_{hy} & & & \\
\end{align*}
\]
Sequence to Sequence

**Many to one**: Encode input sequence in a single vector

**One to many**: Produce output sequence from single input vector
Improved Recurrent Neural Networks

• Long Short-Term Memory Neural Networks (LSTMs)
Topics of Interest

• Reinforcement Learning
  – Learning from experience
  – Example: Learning to levitate

• Self-Supervised Learning
  – Learn a task to learn a feature representation and adapt it to other tasks

• Transformer Language Models (BERT & GPT-2)
• Learning to Learn
• Zero Shot and Few Shot Learning
  – Learn to predict from a few examples
• Attention
• Quantum ML

• Christopher Olah. Understanding LSTM Networks, [http://colah.github.io/posts/2015-08-Understanding-LSTMs/](http://colah.github.io/posts/2015-08-Understanding-LSTMs/)


• Attention
Issues

• Deep Neural Networks are Easily Fooled

• Failures of deep learning

• To understand deep learning we need to understand kernel learning

• Understanding deep learning requires rethinking generalization

• Steps toward deep kernel methods from infinite neural networks

• Do Deep Neural Networks Really Need to be Deep?
• One pixel attack for fooling deep neural networks
  – https://www.youtube.com/watch?v=SA4YEAWVpbk
  – https://github.com/Hyperparticle/one-pixel-attack-keras

• Adversarial Examples that Fool both Computer Vision and Time-Limited Humans
• Alchemy? https://www.youtube.com/watch?v=ORHFOnaEzPc
  – Ali Rahimi
The Future

• AutoML
    • https://github.com/negrinho/deep_architect

• Unsupervised Learning
  – GANs and GAN inspired models
  – Stopping GAN Violence with GUNs
    • https://arxiv.org/abs/1703.02528v1
  – Deep Stubborn Networks
  – Generative Ladder Networks

• Applications of Deep Learning
Tree Based Models

- Tree based methods
  - Simple Tree Classifiers: `DecisionTreeClassifier`
  - Bagging Ensemble: Random Forests: `sklearn.ensemble.RandomForestClassifier`
    - Average output of models
  - Boosting Ensemble: XGBoost
    - Learn a series of weak models
End of Lecture

We want to make a machine that will be proud of us.

- Danny Hillis