Neural Networks

Class based on Raquel Urtasun & Rich Zemel's lectures

Sanja Fidler

University of Toronto

Nov 30, 2016

Let's Revisit High-School Math

- What is the derivative of: $\exp(x)$ wrt x?

Let's Revisit High-School Math

- What is the derivative of: $\exp(x)$ wrt x?
- and

$$\frac{\partial}{\partial x}\left(\frac{1}{1+exp(-x)}\right) \quad ?$$

- What about:

$$\frac{\partial}{\partial x}\max(0,x) \quad ?$$

Let's Revisit High-School Math

- What is the derivative of: $\exp(x)$ wrt x?
- and

$$\frac{\partial}{\partial x}\left(\frac{1}{1+exp(-x)}\right) \quad ?$$

Neural Network

- Neural network is not a religion, or magic
- it is a composite function

Neural Network

- Neural network is not a religion, or magic

Neural Network

- Neural network is not a religion, or magic
- it is a composite function
- but it does leave us praying many times
- and it does work like a charm

Neural Network

- Neural network is not a religion, or magic
- it is a composite function
- but it does leave us praying many times

Today

- Multi-layer Perceptron
- Forward propagation
- Backward propagation

Neural Network

- Neural network is not a religion, or magic
- it is a composite function
- but it does leave us praying many times
- and it does work like a charm or two

Vocabulary of ML

- Data is described as pairs $\mathcal{D} = \{(\mathbf{x}^{(1)}, t^{(1)}), \cdots, (\mathbf{x}^{(N)}, t^{(N)})\}$
  - $\mathbf{x} \in \mathbb{R}^d$ is the input feature vector
  - $t \in \mathbb{R}$ is the target output
  - $^{(i)}$ simply indicates the training examples (we have N in this case)

Motivating Examples

Vocabulary of ML

- Data is described as pairs $\mathcal{D} = \{(\mathbf{x}^{(1)}, t^{(1)}), \cdots, (\mathbf{x}^{(N)}, t^{(N)})\}$
  - $\mathbf{x} \in \mathbb{R}^d$ is the input feature vector
  - $t \in \mathbb{R}$ is the target output
  - $^{(i)}$ simply indicates the training examples (we have N in this case)
- Goal: Learn a mapping from x to y
- If t is continuous, we call it a regression problem
- If t is discrete (categorical), we call it a classification problem

Vocabulary of ML

- Data is described as pairs $\mathcal{D} = \{(\mathbf{x}^{(1)}, t^{(1)}), \cdots, (\mathbf{x}^{(N)}, t^{(N)})\}$
  - $\mathbf{x} \in \mathbb{R}^d$ is the input feature vector
  - $t \in \mathbb{R}$ is the target output
  - $^{(i)}$ simply indicates the training examples (we have N in this case)
- Goal: Learn a mapping from x to y

Vocabulary of ML

- Data is described as pairs $\mathcal{D} = \{(\mathbf{x}^{(1)}, t^{(1)}), \cdots, (\mathbf{x}^{(N)}, t^{(N)})\}$
  - $\mathbf{x} \in \mathbb{R}^d$ is the input feature vector
  - $t \in \mathbb{R}$ is the target output
  - $^{(i)}$ simply indicates the training examples (we have N in this case)
- Goal: Learn a mapping from x to y
- If t is continuous, we call it a regression problem
- If t is discrete (categorical), we call it a classification problem
- Model outputs y, an estimate of t. Simplest form is a linear model:

$$\begin{aligned} \text{Regression:} \quad & y(x) = w_0 + \mathbf{w}^T\mathbf{x} \\ \text{Classification (binary):} \quad & y(x) = \mathrm{sign}(w_0 + \mathbf{w}^T\mathbf{x}) \end{aligned}$$

- Use the training examples to construct hypothesis, or function approximator, that maps $\mathbf{x}$ to predicted y

Vocabulary of ML

- Data is described as pairs $\mathcal{D} = \{(\mathbf{x}^{(1)}, t^{(1)}), \cdots, (\mathbf{x}^{(N)}, t^{(N)})\}$
  - $\mathbf{x} \in \mathbb{R}^d$ is the input feature vector
  - $t \in \mathbb{R}$ is the target output
  - $^{(i)}$ simply indicates the training examples (we have N in this case)
- Goal: Learn a mapping from x to y
- If t is continuous, we call it a regression problem
- If t is discrete (categorical), we call it a classification problem
- Model outputs y, an estimate of t. Simplest form is a linear model:

$$\begin{aligned} \text{Regression:} \quad & y(x) = w_0 + \mathbf{w}^T\mathbf{x} \\ \text{Classification (binary):} \quad & y(x) = \mathrm{sign}(w_0 + \mathbf{w}^T\mathbf{x}) \end{aligned}$$

Regression: Least-Squares Regression

- Define a model

$$y(x) = \text{function}(x, \mathbf{w})$$

Regression: Least-Squares Regression

Regression: Least-Squares Regression

- Define a model

Linear: $y(x) = w_0 + w_1 x$

- Standard loss/cost/objective function measures the squared error between y and the true value t

$$\ell(\mathbf{w}) = \sum_{n=1}^{N} [t^{(n)} - y(x^{(n)})]^2$$

Regression: Least-Squares Regression

- Define a model

Linear: $y(x) = w_0 + w_1 x$

Regression: Least-Squares Regression

- Define a model

$$\text{Linear:} \qquad y(x) = w_0 + w_1 x$$

- Standard loss/cost/objective function measures the squared error between y and the true value t

$$\text{Linear model:} \qquad \ell(\mathbf{w}) = \sum_{n=1}^{N} [t^{(n)} - (w_0 + w_1 x^{(n)})]^2$$

- For a particular hypothesis ($y(x)$ defined by a choice of $\mathbf{w}$, drawn in red), what does the loss represent geometrically?

Regression: Least-Squares Regression

- Define a model

$$\text{Linear:} \qquad y(x) = w_0 + w_1 x$$

- Standard loss/cost/objective function measures the squared error between y and the true value t

$$\text{Linear model:} \qquad \ell(\mathbf{w}) = \sum_{n=1}^{N} [t^{(n)} - (w_0 + w_1 x^{(n)})]^2$$

Regression: Least-Squares Regression

- Define a model

$$\text{Linear:} \qquad y(x) = w_0 + w_1 x$$

- Standard loss/cost/objective function measures the squared error between y and the true value t

$$\text{Linear model:} \qquad \ell(\mathbf{w}) = \sum_{n=1}^{N} [t^{(n)} - (w_0 + w_1 x^{(n)})]^2$$

- How do we obtain weights $\mathbf{w} = (w_0, w_1)$?

Regression: Least-Squares Regression

- Define a model

$$\text{Linear:} \qquad y(x) = w_0 + w_1 x$$

- Standard loss/cost/objective function measures the squared error between y and the true value t

$$\text{Linear model:} \qquad \ell(\mathbf{w}) = \sum_{n=1}^{N} [t^{(n)} - (w_0 + w_1 x^{(n)})]^2$$

- The loss for the red hypothesis is the **sum of the squared vertical errors** (squared lengths of green vertical lines)

Regression: Least-Squares Regression

- Define a model

$$\text{Linear:} \qquad y(x) = w_0 + w_1 x$$

- Standard loss/cost/objective function measures the squared error between y and the true value t

$$\text{Linear model:} \qquad \ell(\mathbf{w}) = \sum_{n=1}^{N} [t^{(n)} - (w_0 + w_1 x^{(n)})]^2$$

- How do we obtain weights $\mathbf{w} = (w_0, w_1)$?
- For the linear model, what kind of a function is $\ell(\mathbf{w})$?

Regression: Least-Squares Regression

- Define a model

$$\text{Linear:} \qquad y(x) = w_0 + w_1 x$$

- Standard loss/cost/objective function measures the squared error between y and the true value t

$$\text{Linear model:} \qquad \ell(\mathbf{w}) = \sum_{n=1}^{N} [t^{(n)} - (w_0 + w_1 x^{(n)})]^2$$

- How do we obtain weights $\mathbf{w} = (w_0, w_1)$? Find $\mathbf{w}$ that minimizes loss $\ell(\mathbf{w})$

Classification: Logistic Regression

- Replace the $sign(\cdot)$ with the sigmoid or logistic function
- We assumed a particular functional form: sigmoid applied to a linear function of the data

$$y(\mathbf{x}) = \sigma\left(\mathbf{w}^T \mathbf{x} + w_0\right)$$

where the sigmoid is defined as

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

Classification: Logistic Regression

- Replace the $sign(\cdot)$ with the sigmoid or logistic function

Classification: Logistic Regression

- Replace the $sign(\cdot)$ with the sigmoid or logistic function
- We assumed a particular functional form: sigmoid applied to a linear function of the data

$$y(\mathbf{x}) = \sigma\left(\mathbf{w}^T\mathbf{x} + w_0\right)$$

where the sigmoid is defined as

$$\sigma(z) = \frac{1}{1+\exp(-z)}$$

- The output is a smooth function of the inputs and the weights. It can be seen as a smoothed and differentiable alternative to $sign(\cdot)$

Classification: Logistic Regression

- Replace the $sign(\cdot)$ with the sigmoid or logistic function
- We assumed a particular functional form: sigmoid applied to a linear function of the data

$$y(\mathbf{x}) = \sigma\left(\mathbf{w}^T\mathbf{x} + w_0\right)$$

where the sigmoid is defined as

$$\sigma(z) = \frac{1}{1+\exp(-z)}$$

Probabilistic Interpretation

- If we have a value between 0 and 1, let's use it to model class probability

$$p(C=0|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + w_0) \quad \text{with} \quad \sigma(z) = \frac{1}{1+\exp(-z)}$$

- Substituting we have

$$p(C=0|\mathbf{x}) = \frac{1}{1+\exp\left(-\mathbf{w}^T\mathbf{x} - w_0\right)}$$

Probabilistic Interpretation

- If we have a value between 0 and 1, let's use it to model class probability

$$p(C=0|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + w_0) \quad \text{with} \quad \sigma(z) = \frac{1}{1+\exp(-z)}$$

Probabilistic Interpretation

- If we have a value between 0 and 1, let's use it to model class probability

$$p(C=0|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + w_0) \quad \text{with} \quad \sigma(z) = \frac{1}{1+\exp(-z)}$$

- Substituting we have

$$p(C=0|\mathbf{x}) = \frac{1}{1+\exp\left(-\mathbf{w}^T\mathbf{x} - w_0\right)}$$

- Suppose we have two classes, how can I compute $p(C=1|\mathbf{x})$?
- Use the marginalization property of probability

$$p(C=1|\mathbf{x}) + p(C=0|\mathbf{x}) = 1$$

Probabilistic Interpretation

- If we have a value between 0 and 1, let's use it to model class probability

$$p(C=0|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + w_0) \quad \text{with} \quad \sigma(z) = \frac{1}{1+\exp(-z)}$$

- Substituting we have

$$p(C=0|\mathbf{x}) = \frac{1}{1+\exp\left(-\mathbf{w}^T\mathbf{x} - w_0\right)}$$

- Suppose we have two classes, how can I compute $p(C=1|\mathbf{x})$?

Decision Boundary for Logistic Regression

- What is the decision boundary for logistic regression?

Probabilistic Interpretation

- If we have a value between 0 and 1, let's use it to model class probability

$$p(C=0|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + w_0) \quad \text{with} \quad \sigma(z) = \frac{1}{1+\exp(-z)}$$

- Substituting we have

$$p(C=0|\mathbf{x}) = \frac{1}{1+\exp\left(-\mathbf{w}^T\mathbf{x} - w_0\right)}$$

- Suppose we have two classes, how can I compute $p(C=1|\mathbf{x})$?
- Use the marginalization property of probability

$$p(C=1|\mathbf{x}) + p(C=0|\mathbf{x}) = 1$$

- Thus:

$$p(C=1|\mathbf{x}) = 1 - \frac{1}{1+\exp\left(-\mathbf{w}^T\mathbf{x} - w_0\right)} = \frac{\exp(-\mathbf{w}^T\mathbf{x} - w_0)}{1+\exp\left(-\mathbf{w}^T\mathbf{x} - w_0\right)}$$

Learning?

- How should we learn the weights $\mathbf{w}, w_0$?

Decision Boundary for Logistic Regression

- What is the decision boundary for logistic regression?
- $p(C = 1|\mathbf{x}, \mathbf{w}) = p(C = 0|\mathbf{x}, \mathbf{w}) = 0.5$
- $p(C = 0|\mathbf{x}, \mathbf{w}) = \sigma\left(\mathbf{w}^T\mathbf{x} + w_0\right) = 0.5$, where $\sigma(z) = \frac{1}{1+\exp(-z)}$
- Decision boundary: $\mathbf{w}^T\mathbf{x} + w_0 = 0$
- Logistic regression has a **linear decision boundary**

Conditional Likelihood

- Assume $t \in \{0, 1\}$, we can write the probability distribution of each of our training points $p(t^{(1)}, \cdots, t^{(N)}|\mathbf{x}^{(1)}, \cdots \mathbf{x}^{(N)}; \mathbf{w})$

Learning?

- How should we learn the weights $\mathbf{w}, w_0$?
- We have a probabilistic model
- Let's use maximum likelihood

(simplify notation: we will write $\mathbf{w}$ to represent both $\mathbf{w}$ and w_0)

Conditional Likelihood

- Assume $t \in \{0, 1\}$, we can write the probability distribution of each of our training points $p(t^{(1)}, \cdots, t^{(N)} | \mathbf{x}^{(1)}, \cdots \mathbf{x}^{(N)}; \mathbf{w})$
- Assuming that the training examples are sampled IID: independent and identically distributed, we can write the *likelihood function*:

$$L(\mathbf{w}) = p(t^{(1)}, \cdots, t^{(N)} | \mathbf{x}^{(1)}, \cdots \mathbf{x}^{(N)}; \mathbf{w}) = \prod_{i=1}^{N} p(t^{(i)} | \mathbf{x}^{(i)}; \mathbf{w})$$

- We can write each probability as (will be useful later):

$$p(t^{(i)} | \mathbf{x}^{(i)}; \mathbf{w}) = p(C = 1 | \mathbf{x}^{(i)}; \mathbf{w})^{t^{(i)}} p(C = 0 | \mathbf{x}^{(i)}; \mathbf{w})^{1 - t^{(i)}}$$

Conditional Likelihood

- Assume $t \in \{0, 1\}$, we can write the probability distribution of each of our training points $p(t^{(1)}, \cdots, t^{(N)} | \mathbf{x}^{(1)}, \cdots \mathbf{x}^{(N)}; \mathbf{w})$
- Assuming that the training examples are sampled IID: independent and identically distributed, we can write the *likelihood function*:

$$L(\mathbf{w}) = p(t^{(1)}, \cdots, t^{(N)} | \mathbf{x}^{(1)}, \cdots \mathbf{x}^{(N)}; \mathbf{w}) = \prod_{i=1}^{N} p(t^{(i)} | \mathbf{x}^{(i)}; \mathbf{w})$$

- We can write each probability as (will be useful later):

$$p(t^{(i)} | \mathbf{x}^{(i)}; \mathbf{w}) = p(C = 1 | \mathbf{x}^{(i)}; \mathbf{w})^{t^{(i)}} p(C = 0 | \mathbf{x}^{(i)}; \mathbf{w})^{1 - t^{(i)}}$$

$$= \left(1 - p(C = 0 | \mathbf{x}^{(i)}; \mathbf{w})\right)^{t^{(i)}} p(C = 0 | \mathbf{x}^{(i)}; \mathbf{w})^{1 - t^{(i)}}$$

- We can learn the model by maximizing the likelihood

$$\max_{\mathbf{w}} L(\mathbf{w}) = \max_{\mathbf{w}} \prod_{i=1}^{N} p(t^{(i)} | \mathbf{x}^{(i)}; \mathbf{w})$$

Conditional Likelihood

- Assume $t \in \{0, 1\}$, we can write the probability distribution of each of our training points $p(t^{(1)}, \cdots, t^{(N)} | \mathbf{x}^{(1)}, \cdots \mathbf{x}^{(N)}; \mathbf{w})$
- Assuming that the training examples are sampled IID: independent and identically distributed, we can write the *likelihood function*:

$$L(\mathbf{w}) = p(t^{(1)}, \cdots, t^{(N)} | \mathbf{x}^{(1)}, \cdots \mathbf{x}^{(N)}; \mathbf{w}) = \prod_{i=1}^{N} p(t^{(i)} | \mathbf{x}^{(i)}; \mathbf{w})$$

- We can write each probability as (will be useful later):

$$p(t^{(i)} | \mathbf{x}^{(i)}; \mathbf{w}) = p(C = 1 | \mathbf{x}^{(i)}; \mathbf{w})^{t^{(i)}} p(C = 0 | \mathbf{x}^{(i)}; \mathbf{w})^{1 - t^{(i)}}$$

$$= \left(1 - p(C = 0 | \mathbf{x}^{(i)}; \mathbf{w})\right)^{t^{(i)}} p(C = 0 | \mathbf{x}^{(i)}; \mathbf{w})^{1 - t^{(i)}}$$

Conditional Likelihood

- Assume $t \in \{0, 1\}$, we can write the probability distribution of each of our training points $p(t^{(1)}, \cdots, t^{(N)} | \mathbf{x}^{(1)}, \cdots \mathbf{x}^{(N)}; \mathbf{w})$
- Assuming that the training examples are sampled IID: independent and identically distributed, we can write the *likelihood function*:

$$L(\mathbf{w}) = p(t^{(1)}, \cdots, t^{(N)} | \mathbf{x}^{(1)}, \cdots \mathbf{x}^{(N)}; \mathbf{w}) = \prod_{i=1}^{N} p(t^{(i)} | \mathbf{x}^{(i)}; \mathbf{w})$$

- We can write each probability as (will be useful later):

$$p(t^{(i)} | \mathbf{x}^{(i)}; \mathbf{w}) = p(C = 1 | \mathbf{x}^{(i)}; \mathbf{w})^{t^{(i)}} p(C = 0 | \mathbf{x}^{(i)}; \mathbf{w})^{1 - t^{(i)}}$$

$$= \left(1 - p(C = 0 | \mathbf{x}^{(i)}; \mathbf{w})\right)^{t^{(i)}} p(C = 0 | \mathbf{x}^{(i)}; \mathbf{w})^{1 - t^{(i)}}$$

Loss Function

$$L(\mathbf{w}) = \prod_{i=1}^{N} p(t^{(i)}|\mathbf{x}^{(i)}) \qquad \text{(likelihood)}$$

Conditional Likelihood

- Assume $t \in \{0, 1\}$, we can write the probability distribution of each of our training points $p(t^{(1)}, \cdots, t^{(N)}|\mathbf{x}^{(1)}, \cdots \mathbf{x}^{(N)}; \mathbf{w})$
- Assuming that the training examples are sampled IID: independent and identically distributed, we can write the *likelihood function*:

$$L(\mathbf{w}) = p(t^{(1)}, \cdots, t^{(N)}|\mathbf{x}^{(1)}, \cdots \mathbf{x}^{(N)}; \mathbf{w}) = \prod_{i=1}^{N} p(t^{(i)}|\mathbf{x}^{(i)}; \mathbf{w})$$

- We can write each probability as (will be useful later):

$$\begin{aligned} p(t^{(i)}|\mathbf{x}^{(i)}; \mathbf{w}) &= p(C = 1|\mathbf{x}^{(i)}; \mathbf{w})^{t^{(i)}} p(C = 0|\mathbf{x}^{(i)}; \mathbf{w})^{1-t^{(i)}} \\ &= \left(1 - p(C = 0|\mathbf{x}^{(i)}; \mathbf{w})\right)^{t^{(i)}} p(C = 0|\mathbf{x}^{(i)}; \mathbf{w})^{1-t^{(i)}} \end{aligned}$$

- We can learn the model by maximizing the likelihood

$$\max_{\mathbf{w}} L(\mathbf{w}) = \max_{\mathbf{w}} \prod_{i=1}^{N} p(t^{(i)}|\mathbf{x}^{(i)}; \mathbf{w})$$

- Easier to maximize the log likelihood $\log L(\mathbf{w})$

Loss Function

$$\begin{aligned} L(\mathbf{w}) &= \prod_{i=1}^{N} p(t^{(i)}|\mathbf{x}^{(i)}) \qquad \text{(likelihood)} \\ &= \prod_{i=1}^{N} \left(1 - p(C = 0|\mathbf{x}^{(i)})\right)^{t^{(i)}} p(C = 0|\mathbf{x}^{(i)})^{1-t^{(i)}} \end{aligned}$$

- We can convert the maximization problem into minimization so that we can write the loss function:

$$\begin{aligned} \ell_{log}(\mathbf{w}) &= -\log L(\mathbf{w}) \\ &= -\sum_{i=1}^{N} \log p(t^{(i)}|\mathbf{x}^{(i)}; \mathbf{w}) \\ &= -\sum_{i=1}^{N} t^{(i)} \log(1 - p(C = 0|\mathbf{x}^{(i)}, \mathbf{w})) - \sum_{i=1}^{N} (1 - t^{(i)}) \log p(C = 0|\mathbf{x}^{(i)}; \mathbf{w}) \end{aligned}$$

Loss Function

$$L(\mathbf{w}) = \prod_{i=1}^{N} p(t^{(i)}|\mathbf{x}^{(i)}) \qquad \text{(likelihood)}$$

- We can convert the maximization problem into minimization so that we can write the loss function:

$$\begin{aligned} \ell_{log}(\mathbf{w}) &= -\log L(\mathbf{w}) \\ &= -\sum_{i=1}^{N} \log p(t^{(i)}|\mathbf{x}^{(i)}; \mathbf{w}) \end{aligned}$$

Loss Function

$$
\begin{aligned}
L(\mathbf{w}) &= \prod_{i=1}^{N} p(t^{(i)}|\mathbf{x}^{(i)}) \qquad \text{(likelihood)} \\
&= \prod_{i=1}^{N} \left(1 - p(C=0|\mathbf{x}^{(i)})\right)^{t^{(i)}} p(C=0|\mathbf{x}^{(i)})^{1-t^{(i)}}
\end{aligned}
$$

- We can convert the maximization problem into minimization so that we can write the loss function:

$$
\begin{aligned}
\ell_{log}(\mathbf{w}) &= -\log L(\mathbf{w}) \\
&= -\sum_{i=1}^{N} \log p(t^{(i)}|\mathbf{x}^{(i)}; \mathbf{w}) \\
&= -\sum_{i=1}^{N} t^{(i)} \log(1 - p(C=0|\mathbf{x}^{(i)}, \mathbf{w})) - \sum_{i=1}^{N} (1 - t^{(i)}) \log p(C=0|\mathbf{x}^{(i)}; \mathbf{w})
\end{aligned}
$$

- Is there a closed form solution?
- It's a convex function of $\mathbf{w}$. Can we get the global optimum?

Loss Function

$$
\begin{aligned}
L(\mathbf{w}) &= \prod_{i=1}^{N} p(t^{(i)}|\mathbf{x}^{(i)}) \qquad \text{(likelihood)} \\
&= \prod_{i=1}^{N} \left(1 - p(C=0|\mathbf{x}^{(i)})\right)^{t^{(i)}} p(C=0|\mathbf{x}^{(i)})^{1-t^{(i)}}
\end{aligned}
$$

- We can convert the maximization problem into minimization so that we can write the loss function:

$$
\begin{aligned}
\ell_{log}(\mathbf{w}) &= -\log L(\mathbf{w}) \\
&= -\sum_{i=1}^{N} \log p(t^{(i)}|\mathbf{x}^{(i)}; \mathbf{w}) \\
&= -\sum_{i=1}^{N} t^{(i)} \log(1 - p(C=0|\mathbf{x}^{(i)}, \mathbf{w})) - \sum_{i=1}^{N} (1 - t^{(i)}) \log p(C=0|\mathbf{x}^{(i)}; \mathbf{w})
\end{aligned}
$$

- Is there a closed form solution?

Gradient Descent

$$
\min_{\mathbf{w}} \ell(\mathbf{w}) = \min_{\mathbf{w}} \left\{ -\sum_{i=1}^{N} t^{(i)} \log(1 - p(C=0|\mathbf{x}^{(i)}, \mathbf{w})) - \sum_{i=1}^{N} (1 - t^{(i)}) \log p(C=0|\mathbf{x}^{(i)}, \mathbf{w}) \right\}
$$

- Gradient descent: iterate and at each iteration compute steepest direction towards optimum, move in that direction, step-size λ

$$
w_j^{(t+1)} \leftarrow w_j^{(t)} - \lambda \frac{\partial \ell(\mathbf{w})}{\partial w_j}
$$

Gradient Descent

$$
\min_{\mathbf{w}} \ell(\mathbf{w}) = \min_{\mathbf{w}} \left\{ -\sum_{i=1}^{N} t^{(i)} \log(1 - p(C=0|\mathbf{x}^{(i)}, \mathbf{w})) - \sum_{i=1}^{N} (1 - t^{(i)}) \log p(C=0|\mathbf{x}^{(i)}, \mathbf{w}) \right\}
$$

Gradient Descent

$$\min_{\mathbf{w}} \ell(\mathbf{w}) = \min_{\mathbf{w}} \left\{ -\sum_{i=1}^{N} t^{(i)} \log(1 - p(C=0|\mathbf{x}^{(i)}, \mathbf{w})) - \sum_{i=1}^{N} (1 - t^{(i)}) \log p(C=0|\mathbf{x}^{(i)}, \mathbf{w}) \right\}$$

- Gradient descent: iterate and at each iteration compute steepest direction towards optimum, move in that direction, step-size λ

$$w_j^{(t+1)} \leftarrow w_j^{(t)} - \lambda \frac{\partial \ell(\mathbf{w})}{\partial w_j}$$

- You can write this in vector form

$$\bigtriangledown \ell(\mathbf{w}) = \left[\frac{\partial \ell(\mathbf{w})}{\partial w_0}, \cdots, \frac{\partial \ell(\mathbf{w})}{\partial w_k} \right]^T, \qquad \text{and} \qquad \triangle(\mathbf{w}) = -\lambda \bigtriangledown \ell(\mathbf{w})$$

Gradient Descent

$$\min_{\mathbf{w}} \ell(\mathbf{w}) = \min_{\mathbf{w}} \left\{ -\sum_{i=1}^{N} t^{(i)} \log(1 - p(C=0|\mathbf{x}^{(i)}, \mathbf{w})) - \sum_{i=1}^{N} (1 - t^{(i)}) \log p(C=0|\mathbf{x}^{(i)}, \mathbf{w}) \right\}$$

- Gradient descent: iterate and at each iteration compute steepest direction towards optimum, move in that direction, step-size λ

$$w_j^{(t+1)} \leftarrow w_j^{(t)} - \lambda \frac{\partial \ell(\mathbf{w})}{\partial w_j}$$

- You can write this in vector form

$$\bigtriangledown \ell(\mathbf{w}) = \left[\frac{\partial \ell(\mathbf{w})}{\partial w_0}, \cdots, \frac{\partial \ell(\mathbf{w})}{\partial w_k} \right]^T,$$

Urtasun, Zemel, Fidler (UofT) Neural Networks Nov 30, 2016 14 / 68

Gradient Descent

$$\min_{\mathbf{w}} \ell(\mathbf{w}) = \min_{\mathbf{w}} \left\{ -\sum_{i=1}^{N} t^{(i)} \log(1 - p(C=0|\mathbf{x}^{(i)}, \mathbf{w})) - \sum_{i=1}^{N} (1 - t^{(i)}) \log p(C=0|\mathbf{x}^{(i)}, \mathbf{w}) \right\}$$

- Gradient descent: iterate and at each iteration compute steepest direction towards optimum, move in that direction, step-size λ

$$w_j^{(t+1)} \leftarrow w_j^{(t)} - \lambda \frac{\partial \ell(\mathbf{w})}{\partial w_j}$$

- You can write this in vector form

$$\bigtriangledown \ell(\mathbf{w}) = \left[\frac{\partial \ell(\mathbf{w})}{\partial w_0}, \cdots, \frac{\partial \ell(\mathbf{w})}{\partial w_k} \right]^T, \qquad \text{and} \qquad \triangle(\mathbf{w}) = -\lambda \bigtriangledown \ell(\mathbf{w})$$

- But where is $\mathbf{w}$?

$$p(C=0|\mathbf{x}) = \frac{1}{1 + \exp\left(-\mathbf{w}^T\mathbf{x} - w_0\right)}$$

Gradient Descent

$$\min_{\mathbf{w}} \ell(\mathbf{w}) = \min_{\mathbf{w}} \left\{ -\sum_{i=1}^{N} t^{(i)} \log(1 - p(C=0|\mathbf{x}^{(i)}, \mathbf{w})) - \sum_{i=1}^{N} (1 - t^{(i)}) \log p(C=0|\mathbf{x}^{(i)}, \mathbf{w}) \right\}$$

- Gradient descent: iterate and at each iteration compute steepest direction towards optimum, move in that direction, step-size λ

$$w_j^{(t+1)} \leftarrow w_j^{(t)} - \lambda \frac{\partial \ell(\mathbf{w})}{\partial w_j}$$

- You can write this in vector form

$$\bigtriangledown \ell(\mathbf{w}) = \left[\frac{\partial \ell(\mathbf{w})}{\partial w_0}, \cdots, \frac{\partial \ell(\mathbf{w})}{\partial w_k} \right]^T, \qquad \text{and} \qquad \triangle(\mathbf{w}) = -\lambda \bigtriangledown \ell(\mathbf{w})$$

- But where is $\mathbf{w}$?

Urtasun, Zemel, Fidler (UofT) Neural Networks Nov 30, 2016 14 / 68

Multi-class Logistic Regression

- What if we have more classes (not just 2)?

Gradient Descent

$$\min_{\mathbf{w}} \ell(\mathbf{w}) = \min_{\mathbf{w}} \left\{ -\sum_{i=1}^{N} t^{(i)} \log(1 - p(C=0|\mathbf{x}^{(i)}, \mathbf{w})) - \sum_{i=1}^{N} (1 - t^{(i)}) \log p(C=0|\mathbf{x}^{(i)}, \mathbf{w}) \right\}$$

- Gradient descent: iterate and at each iteration compute steepest direction towards optimum, move in that direction, step-size λ

$$w_j^{(t+1)} \leftarrow w_j^{(t)} - \lambda \frac{\partial \ell(\mathbf{w})}{\partial w_j}$$

- You can write this in vector form

$$\bigtriangledown \ell(\mathbf{w}) = \left[\frac{\partial \ell(\mathbf{w})}{\partial w_0}, \cdots, \frac{\partial \ell(\mathbf{w})}{\partial w_k} \right]^T, \qquad \text{and} \qquad \triangle(\mathbf{w}) = -\lambda \bigtriangledown \ell(\mathbf{w})$$

- But where is $\mathbf{w}$?

$$p(C=0|\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T\mathbf{x} - w_0)}, \quad p(C=1|\mathbf{x}) = \frac{\exp(-\mathbf{w}^T\mathbf{x} - w_0)}{1 + \exp(-\mathbf{w}^T\mathbf{x} - w_0)}$$

- Derive the updates as an exercise!

Multi-class Logistic Regression

- The likelihood

$$p(\mathbf{T}|\mathbf{w}_1, \cdots, \mathbf{w}_k) = \prod_{n=1}^{N} \prod_{k=1}^{K} p(C_k|\mathbf{x}^{(n)})^{t_k^{(n)}} = \prod_{n=1}^{N} \prod_{k=1}^{K} y_k^{(n)}(\mathbf{x}^{(n)})^{t_k^{(n)}}$$

with

$$p(C_k|\mathbf{x}) = y_k(\mathbf{x}) = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

where n-th row of $\mathbf{T}$ is 1-of-K encoding of example n and

$$z_k = \mathbf{w}_k^T \mathbf{x} + w_{k0}$$

Multi-class Logistic Regression

- What if we have more classes (not just 2)?
- Associate a set of weights with each class, then use a normalized exponential output

$$p(C_k|\mathbf{x}) = y_k(\mathbf{x}) = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

where the **activations** are given by

$$z_k = \mathbf{w}_k^T \mathbf{x}$$

- The function $\frac{\exp(z_k)}{\sum_j \exp(z_j)}$ is called a **softmax function**

Multi-class Logistic Regression

- The likelihood

$$p(\mathbf{T}|\mathbf{w}_1, \cdots, \mathbf{w}_k) = \prod_{n=1}^{N}\prod_{k=1}^{K} p(C_k|\mathbf{x}^{(n)})^{t_k^{(n)}} = \prod_{n=1}^{N}\prod_{k=1}^{K} y_k^{(n)}(\mathbf{x}^{(n)})^{t_k^{(n)}}$$

with

$$p(C_k|\mathbf{x}) = y_k(\mathbf{x}) = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

where n-th row of $\mathbf{T}$ is 1-of-K encoding of example n and

$$z_k = \mathbf{w}_k^T\mathbf{x} + w_{k0}$$

- Derive the loss by computing the negative log-likelihood:

$$E(\mathbf{w}_1, \cdots, \mathbf{w}_K) = -\log p(\mathbf{T}|\mathbf{w}_1, \cdots, \mathbf{w}_K) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)} \log[y_k^{(n)}(\mathbf{x}^{(n)})]$$

This is known as the **cross-entropy** error for multiclass classification

- How do we obtain the weights?

Multi-class Logistic Regression

- The likelihood

$$p(\mathbf{T}|\mathbf{w}_1, \cdots, \mathbf{w}_k) = \prod_{n=1}^{N}\prod_{k=1}^{K} p(C_k|\mathbf{x}^{(n)})^{t_k^{(n)}} = \prod_{n=1}^{N}\prod_{k=1}^{K} y_k^{(n)}(\mathbf{x}^{(n)})^{t_k^{(n)}}$$

with

$$p(C_k|\mathbf{x}) = y_k(\mathbf{x}) = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

where n-th row of $\mathbf{T}$ is 1-of-K encoding of example n and

$$z_k = \mathbf{w}_k^T\mathbf{x} + w_{k0}$$

- Derive the loss by computing the negative log-likelihood:

$$E(\mathbf{w}_1, \cdots, \mathbf{w}_K) = -\log p(\mathbf{T}|\mathbf{w}_1, \cdots, \mathbf{w}_K) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)} \log[y_k^{(n)}(\mathbf{x}^{(n)})]$$

This is known as the **cross-entropy** error for multiclass classification

Training Multi-class Logistic Regression

- How do we obtain the weights?

$$E(\mathbf{w}_1, \cdots, \mathbf{w}_K) = -\log p(\mathbf{T}|\mathbf{w}_1, \cdots, \mathbf{w}_K) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)} \log[y_k^{(n)}(\mathbf{x}^{(n)})]$$

- Do gradient descent, where the derivatives are

$$\frac{\partial y_j^{(n)}}{\partial z_k^{(n)}} = \delta(k,j)y_j^{(n)} - y_j^{(n)}y_k^{(n)}$$

and

$$\frac{\partial E}{\partial z_k^{(n)}} = \sum_{j=1}^{K} \frac{\partial E}{\partial y_j^{(n)}} \cdot \frac{\partial y_j^{(n)}}{\partial z_k^{(n)}} = y_k^{(n)} - t_k^{(n)}$$

$$\frac{\partial E}{\partial w_{k,j}} = \sum_{n=1}^{N}\sum_{j=1}^{K} \frac{\partial E}{\partial y_j^{(n)}} \cdot \frac{\partial y_j^{(n)}}{\partial z_k^{(n)}} \cdot \frac{\partial z_k^{(n)}}{\partial w_{k,j}} = \sum_{n=1}^{N}(y_k^{(n)} - t_k^{(n)}) \cdot x_j^{(n)}$$

- The derivative is the error times the input

Limitations of Linear Classifiers

- Linear classifiers (e.g., logistic regression) classify inputs based on linear combinations of features x_i

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?

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

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?

- The positive and negative cases cannot be separated by a plane
- What can we do?

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?

- The positive and negative cases cannot be separated by a plane

How to Construct Nonlinear Classifiers?

- We would like to construct non-linear discriminative classifiers that utilize functions of input variables
- Use a large number of simpler functions

How to Construct Nonlinear Classifiers?

- We would like to construct non-linear discriminative classifiers that utilize functions of input variables

Fitting a Polynomial

- What if our linear model is not good? How can we create a more complicated model?
- We can create a more complicated model by defining input variables that are combinations of components of $\mathbf{x}$

How to Construct Nonlinear Classifiers?

- We would like to construct non-linear discriminative classifiers that utilize functions of input variables
- Use a large number of simpler functions
  - If these functions are fixed (Gaussian, sigmoid, polynomial basis functions), then optimization still involves linear combinations of (fixed functions of) the inputs

Fitting a Polynomial

- What if our linear model is not good? How can we create a more complicated model?
- We can create a more complicated model by defining input variables that are combinations of components of $\mathbf{x}$
- Example: an M-th order polynomial function of one dimensional feature x:

$$y(x, \mathbf{w}) = w_0 + \sum_{j=1}^{M} w_j x^j$$

where x^j is the j-th power of x

- We can use the same approach to optimize for the weights $\mathbf{w}$

Fitting a Polynomial

- What if our linear model is not good? How can we create a more complicated model?
- We can create a more complicated model by defining input variables that are combinations of components of $\mathbf{x}$
- Example: an M-th order polynomial function of one dimensional feature x:

$$y(x, \mathbf{w}) = w_0 + \sum_{j=1}^{M} w_j x^j$$

where x^j is the j-th power of x

Which Fit is Best?

How to Construct Nonlinear Classifiers?

- We would like to construct non-linear discriminative classifiers that utilize functions of input variables
- Use a large number of simpler functions
  - If these functions are fixed (Gaussian, sigmoid, polynomial basis functions), then optimization still involves linear combinations of (fixed functions of) the inputs

Inspiration: The Brain

- Many machine learning methods inspired by biology, eg the (human) brain
- Our brain has $\sim 10^{11}$ neurons, each of which communicates (is connected) to $\sim 10^4$ other neurons

Figure: The basic computational unit of the brain: Neuron

[Pic credit: http://cs231n.github.io/neural-networks-1/]

How to Construct Nonlinear Classifiers?

- We would like to construct non-linear discriminative classifiers that utilize functions of input variables
- Use a large number of simpler 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 $\rightarrow$ need an efficient method of training extra parameters

Mathematical Model of a Neuron

- Neural networks define functions of the inputs (hidden features), computed by neurons
- Artificial neurons are called units

Figure: A mathematical model of the neuron in a neural network

[Pic credit: http://cs231n.github.io/neural-networks-1/]

Activation Functions

Most commonly used activation functions:

- Sigmoid: $\sigma(z) = \frac{1}{1+\exp(-z)}$
- Tanh: $\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$
- ReLU (Rectified Linear Unit): $\mathrm{ReLU}(z) = \max(0, z)$

Neural Network Architecture (Multi-Layer Perceptron)

- Network with one layer of four hidden units:

Figure: Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

- Each unit computes its value based on linear combination of values of units that point into it, and an activation function

[http://cs231n.github.io/neural-networks-1/]

Neuron in Python

- Example in Python of a neuron with a sigmoid activation function

```
class Neuron(object):
  # ...
  def forward(inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function
    return firing_rate
```

Figure: Example code for computing the activation of a single neuron

[http://cs231n.github.io/neural-networks-1/]

Neural Network Architecture (Multi-Layer Perceptron)

- Going deeper: a 3-layer neural network with two layers of hidden units

Figure: A 3-layer neural net with 3 input units, 4 hidden units in the first and second hidden layer and 1 output unit

- Naming conventions; a N-layer neural network:
  - $N-1$ layers of hidden units
  - One output layer

[http://cs231n.github.io/neural-networks-1/]

Neural Network Architecture (Multi-Layer Perceptron)

- Network with one layer of four hidden units:

Figure: Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

- Naming conventions; a 2-layer neural network:
  - One layer of hidden units
  - One output layer
    (we do not count the inputs as a layer)

[http://cs231n.github.io/neural-networks-1/]

Representational Power

- Neural network with at **least one hidden layer** is a universal approximator (can represent any function).
  Proof in: Approximation by Superpositions of Sigmoidal Function, Cybenko, paper

- The capacity of the network increases with more hidden units and more hidden layers

Representational Power

- Neural network with at **least one hidden layer** is a universal approximator (can represent any function).
  Proof in: Approximation by Superpositions of Sigmoidal Function, Cybenko, paper

Neural Networks

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

Representational Power

- Neural network with at **least one hidden layer** is a universal approximator (can represent any function).
  Proof in: Approximation by Superpositions of Sigmoidal Function, Cybenko, paper

- The capacity of the network increases with more hidden units and more hidden layers
- Why go deeper? Read eg: Do Deep Nets Really Need to be Deep? Jimmy Ba, Rich Caruana, Paper: paper]

[http://cs231n.github.io/neural-networks-1/]

Forward Pass: What does the Network Compute?

- Output of the network can be written as:

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

Forward Pass: What does the Network Compute?

Forward Pass: What does the Network Compute?

- Output of the network can be written as:

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

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

(j indexing hidden units, k indexing the output units, D number of inputs)

- Activation functions f, g: sigmoid/logistic, tanh, or rectified linear (ReLU)

$$\sigma(z) = \frac{1}{1+\exp(-z)}, \quad \tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}, \quad \mathrm{ReLU}(z) = \max(0, z)$$

Forward Pass: What does the Network Compute?

- Output of the network can be written as:

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

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

(j indexing hidden units, k indexing the output units, D number of inputs)

Forward Pass in Python

- Example code for a forward pass for a 3-layer network in Python:

```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

- Can be implemented efficiently using matrix operations
- Example above: W_1 is matrix of size 4×3, W_2 is 4×4. What about biases and W_3?

[http://cs231n.github.io/neural-networks-1/]

Forward Pass in Python

- Example code for a forward pass for a 3-layer network in Python:

```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

- Can be implemented efficiently using matrix operations

Special Case

- What is a single layer (no hiddens) network with a sigmoid act. function?

- Network:

$$o_k(\mathbf{x}) = \frac{1}{1+\exp(-z_k)}$$

$$z_k = w_{k0} + \sum_{j=1}^{J} x_j w_{kj}$$

Special Case

- What is a single layer (no hiddens) network with a sigmoid act. function?

Example Application

- Classify image of handwritten digit (32x32 pixels): 4 vs non-4

Special Case

- What is a single layer (no hiddens) network with a sigmoid act. function?

- Network:

$$o_k(\mathbf{x}) = \frac{1}{1+\exp(-z_k)}$$
$$z_k = w_{k0} + \sum_{j=1}^{J} x_j w_{kj}$$

- Logistic regression!

Example Application

- Classify image of handwritten digit (32x32 pixels): 4 vs non-4

- How would you build your network?
- For example, use one hidden layer and the sigmoid activation function:

$$o_k(\mathbf{x}) = \frac{1}{1+\exp(-z_k)}$$
$$z_k = w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj}$$

Example Application

- Classify image of handwritten digit (32x32 pixels): 4 vs non-4

- How would you build your network?

Training Neural Networks

- Find weights:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} \operatorname{loss}(\mathbf{o}^{(n)}, \mathbf{t}^{(n)})$$

where $\mathbf{o} = f(\mathbf{x}; \mathbf{w})$ is the output of a neural network, $\mathbf{t}$ is ground-truth

Example Application

- Classify image of handwritten digit (32x32 pixels): 4 vs non-4

- How would you build your network?
- For example, use one hidden layer and the sigmoid activation function:

$$\begin{aligned} o_k(\mathbf{x}) &= \frac{1}{1+\exp(-z_k)} \\ z_k &= w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj} \end{aligned}$$

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

Training Neural Networks

- Find weights:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} \operatorname{loss}(\mathbf{o}^{(n)}, \mathbf{t}^{(n)})$$

where $\mathbf{o} = f(\mathbf{x}; \mathbf{w})$ is the output of a neural network, $\mathbf{t}$ is ground-truth

- Define a loss function, eg:
  - Squared loss: $\sum_k \frac{1}{2}(o_k^{(n)} - t_k^{(n)})^2$
  - Cross-entropy loss: $-\sum_k t_k^{(n)} \log o_k^{(n)}$
- Gradient descent:

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{\partial E}{\partial \mathbf{w}^t}$$

where η is the learning rate (and E is error/loss)

Training Neural Networks

- Find weights:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} \operatorname{loss}(\mathbf{o}^{(n)}, \mathbf{t}^{(n)})$$

where $\mathbf{o} = f(\mathbf{x}; \mathbf{w})$ is the output of a neural network, $\mathbf{t}$ is ground-truth

- Define a loss function, eg:
  - Squared loss: $\sum_k \frac{1}{2}(o_k^{(n)} - t_k^{(n)})^2$
  - Cross-entropy loss: $-\sum_k t_k^{(n)} \log o_k^{(n)}$

Training Neural Networks: Back-propagation

- Back-propagation: an efficient method for computing gradients needed to perform gradient-based optimization of the weights in a multi-layer network

Useful Derivatives

name	function	derivative
Sigmoid	$\sigma(z) = \frac{1}{1+\exp(-z)}$	$\sigma(z) \cdot (1 - \sigma(z))$
Tanh	$\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$	$1/\cosh^2(z)$
ReLU	$\mathrm{ReLU}(z) = \max(0, z)$	$\begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases}$

Training Neural Networks: Back-propagation

- Back-propagation: an efficient method for computing gradients needed to perform gradient-based optimization of the weights in a multi-layer network

Training neural nets:

Loop until convergence:

- for each example n
  1. Given input $\mathbf{x}^{(n)}$, propagate activity forward ($\mathbf{x}^{(n)} \to \mathbf{h}^{(n)} \to o^{(n)}$) (**forward pass**)
  2. Propagate gradients backward (**backward pass**)
  3. Update each weight (via gradient descent)

- Given any error function E, activation functions $g()$ and $f()$, just need to derive gradients

Training Neural Networks: Back-propagation

- Back-propagation: an efficient method for computing gradients needed to perform gradient-based optimization of the weights in a multi-layer network

Training neural nets:

Loop until convergence:

- for each example n
  1. Given input $\mathbf{x}^{(n)}$, propagate activity forward ($\mathbf{x}^{(n)} \to \mathbf{h}^{(n)} \to o^{(n)}$) (**forward pass**)
  2. Propagate gradients backward (**backward pass**)
  3. Update each weight (via gradient descent)

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

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

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

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

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, its easy to get the error derivatives for the weights going into a hidden unit
- This is just the chain rule!

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, its easy to get the error derivatives for the weights going into a hidden unit

Computing Gradients: Single Layer Network

- Let's take a single layer network and draw it a bit differently

Computing Gradients: Single Layer Network

- Let's take a single layer network

Computing Gradients: Single Layer Network

- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}}$$

Computing Gradients: Single Layer Network

- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} =$$

Computing Gradients: Single Layer Network

$$\delta_k^o = \frac{\partial E}{\partial o_k}$$

- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \underbrace{\frac{\partial E}{\partial o_k}}_{\delta_k^o}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}}$$

Computing Gradients: 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

Computing Gradients: Single Layer Network

$$\delta_k^z = \delta_k^o \cdot \frac{\partial o_k}{\partial z_k}$$

- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}} = \underbrace{\delta_k^o \cdot \frac{\partial o_k}{\partial z_k}}_{\delta_k^z}\frac{\partial z_k}{\partial w_{ki}}$$

Computing Gradients: Single Layer Network

$$\frac{\partial o_k}{\partial z_k}$$

- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}} = \delta_k^o \frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}}$$

Gradient Descent for Single Layer Network

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

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

Using logistic activation functions:

$$\begin{aligned} o_k^{(n)} &= g(z_k^{(n)}) = (1+\exp(-z_k^{(n)}))^{-1} \\ \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} &= o_k^{(n)}(1-o_k^{(n)}) \end{aligned}$$

Computing Gradients: Single Layer Network

$$\frac{\partial z_k}{\partial w_{ki}}$$

- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}} = \delta_k^z \frac{\partial z_k}{\partial w_{ki}} = \delta_k^z \cdot x_i$$

Gradient Descent for Single Layer Network

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

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

Using logistic activation functions:

$$\begin{aligned} o_k^{(n)} &= g(z_k^{(n)}) = (1+\exp(-z_k^{(n)}))^{-1} \\ \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} &= o_k^{(n)}(1-o_k^{(n)}) \end{aligned}$$

- The error gradient is then:

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

Gradient Descent for Single Layer Network

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

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

Using logistic activation functions:

$$\begin{aligned} o_k^{(n)} &= g(z_k^{(n)}) = (1+\exp(-z_k^{(n)}))^{-1} \\ \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} &= o_k^{(n)}(1-o_k^{(n)}) \end{aligned}$$

- The error gradient is then:

$$\frac{\partial E}{\partial w_{ki}} =$$

Urtasun, Zemel, Fidler (UofT) Neural Networks Nov 30, 2016 46 / 68

Gradient Descent for Single Layer Network

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

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

Using logistic activation functions:

$$\begin{aligned} o_k^{(n)} &= g(z_k^{(n)}) = (1+\exp(-z_k^{(n)}))^{-1} \\ \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} &= o_k^{(n)}(1-o_k^{(n)}) \end{aligned}$$

- 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:

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} =$$

Gradient Descent for Single Layer Network

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

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

Using logistic activation functions:

$$\begin{aligned} o_k^{(n)} &= g(z_k^{(n)}) = (1+\exp(-z_k^{(n)}))^{-1} \\ \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} &= o_k^{(n)}(1-o_k^{(n)}) \end{aligned}$$

- The error gradient is then:

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

Urtasun, Zemel, Fidler (UofT) Neural Networks Nov 30, 2016 46 / 68

Multi-layer Neural Network

Gradient Descent for Single Layer Network

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

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

Using logistic activation functions:

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

- 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:

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

Back-propagation: Sketch on One Training Case

- Convert discrepancy between each output and its target value into an error derivative

$$E = \frac{1}{2} \sum_k (o_k - t_k)^2; \qquad \frac{\partial E}{\partial o_k} = o_k - t_k$$

- 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})]

Back-propagation: Sketch on One Training Case

- Convert discrepancy between each output and its target value into an error derivative

$$E = \frac{1}{2} \sum_k (o_k - t_k)^2; \qquad \frac{\partial E}{\partial o_k} = o_k - t_k$$

Gradient Descent for Multi-layer Network

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

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

Back-propagation: Sketch on One Training Case

- Convert discrepancy between each output and its target value into an error derivative

$$E = \frac{1}{2}\sum_k (o_k - t_k)^2; \qquad \frac{\partial E}{\partial o_k} = o_k - t_k$$

- 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

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

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

$$\frac{\partial E}{\partial h_j^{(n)}} = \sum_k \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial h_j^{(n)}} =$$

Gradient Descent for Multi-layer Network

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

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

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

$$\frac{\partial E}{\partial h_j^{(n)}} =$$

Gradient Descent for Multi-layer Network

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

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

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

$$\frac{\partial E}{\partial h_j^{(n)}} = \sum_k \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial h_j^{(n)}} = \sum_k \delta_k^{z,(n)} w_{kj} := \delta_j^{h,(n)}$$

$$\frac{\partial E}{\partial v_{ji}} = \sum_{n=1}^{N} \frac{\partial E}{\partial h_j^{(n)}} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} \frac{\partial u_j^{(n)}}{\partial v_{ji}} =$$

Gradient Descent for Multi-layer Network

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

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

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

$$\frac{\partial E}{\partial h_j^{(n)}} = \sum_k \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial h_j^{(n)}} = \sum_k \delta_k^{z,(n)} w_{kj} := \delta_j^{h,(n)}$$

Gradient Descent for Multi-layer Network

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

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

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

$$\frac{\partial E}{\partial h_j^{(n)}} = \sum_k \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial h_j^{(n)}} = \sum_k \delta_k^{z,(n)} w_{kj} := \delta_j^{h,(n)}$$

$$\frac{\partial E}{\partial v_{ji}} = \sum_{n=1}^{N} \frac{\partial E}{\partial h_j^{(n)}} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \delta_j^{h,(n)} f'(u_j^{(n)}) \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \delta_j^{u,(n)} x_i^{(n)}$$

Gradient Descent for Multi-layer Network

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

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

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

$$\frac{\partial E}{\partial h_j^{(n)}} = \sum_k \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial h_j^{(n)}} = \sum_k \delta_k^{z,(n)} w_{kj} := \delta_j^{h,(n)}$$

$$\frac{\partial E}{\partial v_{ji}} = \sum_{n=1}^{N} \frac{\partial E}{\partial h_j^{(n)}} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \delta_j^{h,(n)} f'(u_j^{(n)}) \frac{\partial u_j^{(n)}}{\partial v_{ji}} =$$

Choosing Activation and Loss Functions

- When using a neural network for regression?

Toy Code (Matlab): Neural Net Trainer

```
% 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
```

This code has a few bugs with indices...

28

Ranzato

Choosing Activation and Loss Functions

- When using a neural network for regression? Sigmoid activation (but not at the output layer) and MSE as the loss function work well
- For classification, if it is a binary (2-class) problem?

Choosing Activation and Loss Functions

- When using a neural network for regression? Sigmoid activation (but not at the output layer) and MSE as the loss function work well

Choosing Activation and Loss Functions

- When using a neural network for regression? Sigmoid activation (but not at the output layer) and MSE as the loss function work well
- For classification, if it is a binary (2-class) problem? Binary cross-entropy error function often does better

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

$$o^{(n)} = (1 + \exp(-z^{(n)})^{-1}$$

- For multi-class classification?

Choosing Activation and Loss Functions

- When using a neural network for regression? Sigmoid activation (but not at the output layer) and MSE as the loss function work well
- For classification, if it is a binary (2-class) problem? Binary cross-entropy error function often does better

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

$$o^{(n)} = (1 + \exp(-z^{(n)})^{-1}$$

Example Application

- Now trying to classify image of handwritten digit: 32x32 pixels
- 10 output units, 1 per digit
- Use the softmax function:

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

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

- What is J ?

Choosing Activation and Loss Functions

- When using a neural network for regression? Sigmoid activation (but not at the output layer) and MSE as the loss function work well
- For classification, if it is a binary (2-class) problem? Binary cross-entropy error function often does better

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

$$o^{(n)} = (1 + \exp(-z^{(n)})^{-1}$$

- For multi-class classification? Typically softmax at the output layer, and cross-entropy loss:

$$E = -\sum_{n} \sum_{k} t_k^{(n)} \log o_k^{(n)}$$

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

Ways to Use Weight Derivatives

- How often to update
  - after a full sweep through the training data (batch gradient descent)

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} \frac{\partial E(\mathbf{o}^{(n)}, \mathbf{t}^{(n)}; \mathbf{w})}{\partial w_{ki}}$$

Ways to Use Weight Derivatives

- How often to update

Ways to Use Weight Derivatives

- How often to update
  - after a full sweep through the training data (batch gradient descent)

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} \frac{\partial E(\mathbf{o}^{(n)}, \mathbf{t}^{(n)}; \mathbf{w})}{\partial w_{ki}}$$

  - after each training case (stochastic gradient descent)
  - after a mini-batch of training cases

Ways to Use Weight Derivatives

- How often to update
  - after a full sweep through the training data (batch gradient descent)

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} \frac{\partial E(\mathbf{o}^{(n)}, \mathbf{t}^{(n)}; \mathbf{w})}{\partial w_{ki}}$$

  - after each training case (stochastic gradient descent)

Ways to Use Weight Derivatives

- How often to update
  - after a full sweep through the training data (batch gradient descent)

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} \frac{\partial E(\mathbf{o}^{(n)}, \mathbf{t}^{(n)}; \mathbf{w})}{\partial w_{ki}}$$

  - after each training case (stochastic gradient descent)
  - after a mini-batch of training cases
- How much to update
  - Use a fixed learning rate

Ways to Use Weight Derivatives

- How often to update
  - after a full sweep through the training data (batch gradient descent)

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} \frac{\partial E(\mathbf{o}^{(n)}, \mathbf{t}^{(n)}; \mathbf{w})}{\partial w_{ki}}$$

  - after each training case (stochastic gradient descent)
  - after a mini-batch of training cases
- How much to update

Ways to Use Weight Derivatives

- How often to update
  - after a full sweep through the training data (batch gradient descent)

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} \frac{\partial E(\mathbf{o}^{(n)}, \mathbf{t}^{(n)}; \mathbf{w})}{\partial w_{ki}}$$

  - after each training case (stochastic gradient descent)
  - after a mini-batch of training cases
- How much to update
  - Use a fixed learning rate
  - Adapt the learning rate
  - Add momentum

$$\begin{aligned} w_{ki} &\leftarrow w_{ki} - v \\ v &\leftarrow \gamma v + \eta \frac{\partial E}{\partial w_{ki}} \end{aligned}$$

Ways to Use Weight Derivatives

- How often to update
  - after a full sweep through the training data (batch gradient descent)

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} \frac{\partial E(\mathbf{o}^{(n)}, \mathbf{t}^{(n)}; \mathbf{w})}{\partial w_{ki}}$$

  - after each training case (stochastic gradient descent)
  - after a mini-batch of training cases
- How much to update
  - Use a fixed learning rate
  - Adapt the learning rate

Monitor Loss During Training

- Check how your loss behaves during training, to spot wrong hyperparameters, bugs, etc

Figure: **Left:** Good vs bad parameter choices, **Right:** How a real loss might look like during training. What are the bumps caused by? How could we get a more smooth loss?

Comparing Optimization Methods

[http://cs231n.github.io/neural-networks-3/, Alec Radford]

Overfitting

- The training data contains information about the regularities in the mapping from input to output. But it also contains noise
  - The target values may be unreliable.
  - There is sampling error: There will be accidental regularities just because of the particular training cases that were chosen

Monitor Accuracy on Train/Validation During Training

- Check how your desired performance metrics behaves during training

[http://cs231n.github.io/neural-networks-3/]

Preventing Overfitting

- Use a model that has the right capacity:
  - enough to model the true regularities
  - not enough to also model the spurious regularities (assuming they are weaker)

Overfitting

- The training data contains information about the regularities in the mapping from input to output. But it also contains noise
  - The target values may be unreliable.
  - There is sampling error: There will be accidental regularities just because of the particular training cases that were chosen
- When we fit the model, it cannot tell which regularities are real and which are caused by sampling error.
  - So it fits both kinds of regularity.
  - If the model is very flexible it can model the sampling error really well. This is a disaster.

Limiting the size of the Weights

- Weight-decay involves adding an extra term to the cost function that penalizes the squared weights.

$$C = \ell + \frac{\lambda}{2} \sum_i w_i^2$$

- Keeps weights small unless they have big error derivatives.

Preventing Overfitting

- Use a model that has the right capacity:
  - enough to model the true regularities
  - not enough to also model the spurious regularities (assuming they are weaker)
- Standard ways to limit the capacity of a neural net:
  - Limit the number of hidden units.
  - Limit the norm of the weights.
  - Stop the learning before it has time to overfit.

The Effect of Weight-decay

- It prevents the network from using weights that it does not need
  - This can often improve generalization a lot.
  - It helps to stop it from fitting the sampling error.
  - It makes a smoother model in which the output changes more slowly as the input changes.
- But, if the network has two very similar inputs it prefers to put half the weight on each rather than all the weight on one → other form of weight decay?

The Effect of Weight-decay

- It prevents the network from using weights that it does not need
  - This can often improve generalization a lot.
  - It helps to stop it from fitting the sampling error.
  - It makes a smoother model in which the output changes more slowly as the input changes.

Deciding How Much to Restrict the Capacity

- How do we decide which regularizer to use and how strong to make it?
- So use a separate validation set to do model selection.

Deciding How Much to Restrict the Capacity

- How do we decide which regularizer to use and how strong to make it?

Preventing Overfitting by Early Stopping

- If we have lots of data and a big model, its very expensive to keep re-training it with different amounts of weight decay

Using a Validation Set

- Divide the total dataset into three subsets:
  - Training data is used for learning the parameters of the model.
  - Validation data is not used for learning but is used for deciding what type of model and what amount of regularization works best
  - Test data is used to get a final, unbiased estimate of how well the network works. We expect this estimate to be worse than on the validation data
- We could then re-divide the total dataset to get another unbiased estimate of the true error rate.

Preventing Overfitting by Early Stopping

- If we have lots of data and a big model, its very expensive to keep re-training it with different amounts of weight decay
- It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse
- The capacity of the model is limited because the weights have not had time to grow big.

Preventing Overfitting by Early Stopping

- If we have lots of data and a big model, its very expensive to keep re-training it with different amounts of weight decay
- It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse

Dropout

(a) Standard Neural Net (b) After applying dropout.

- During training, dropout only keeps a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise.

[Srivastava et al., Dropout: A Simple Way to Prevent Neural Networks from Overfitting, 2014]

Why Early Stopping Works

- When the weights are very small, every hidden unit is in its linear range.
  - So a net with a large layer of hidden units is linear.
  - It has no more capacity than a linear net in which the inputs are directly connected to the outputs!
- As the weights grow, the hidden units start using their non-linear ranges so the capacity grows.

Today

- Deep learning for Object Recognition

Neural Networks II

Sanja Fidler

Some slides credited to Raquel Urtasun, Rich Zemel

University of Toronto

Nov 30, 2016

Neural Nets for Object Recognition

- People are very good at recognizing shapes
  - Intrinsically difficult, computers are bad at it

Neural Nets for Object Recognition

- People are very good at recognizing shapes

Why is it a Problem?

- Difficult scene conditions

[From: Grauman & Leibe]

Neural Nets for Object Recognition

- People are very good at recognizing shapes
  - Intrinsically difficult, computers are bad at it
- Why is it difficult?

Why is it a Problem?

- Tones of classes

[Biederman]

Why is it a Problem?

- Huge within-class variations. Recognition is mainly about modeling variation.

[Pic from: S. Lazebnik]

Neural Nets for Object Recognition

- People are very good at recognizing shapes
  - Intrinsically difficult, computers are bad at it
- Some reasons why it is difficult:
  - Segmentation: Real scenes are cluttered

Neural Nets for Object Recognition

- People are very good at recognizing shapes
  - Intrinsically difficult, computers are bad at it
- Some reasons why it is difficult:

Neural Nets for Object Recognition

- People are very good at recognizing shapes
  - Intrinsically difficult, computers are bad at it
- Some reasons why it is difficult:
  - Segmentation: Real scenes are cluttered
  - Invariances: We are very good at ignoring all sorts of variations that do not affect shape
  - Deformations: Natural shape classes allow variations (faces, letters, chairs)

Neural Nets for Object Recognition

- People are very good at recognizing shapes
  - Intrinsically difficult, computers are bad at it
- Some reasons why it is difficult:
  - Segmentation: Real scenes are cluttered
  - Invariances: We are very good at ignoring all sorts of variations that do not affect shape

How to Deal with Large Input Spaces

- How can we apply neural nets to images?

Neural Nets for Object Recognition

- People are very good at recognizing shapes
  - Intrinsically difficult, computers are bad at it
- Some reasons why it is difficult:
  - Segmentation: Real scenes are cluttered
  - Invariances: We are very good at ignoring all sorts of variations that do not affect shape
  - Deformations: Natural shape classes allow variations (faces, letters, chairs)
  - A huge amount of computation is required

How to Deal with Large Input Spaces

- How can we apply neural nets to images?
- Images can have millions of pixels, i.e., $\mathbf{x}$ is very high dimensional
- How many parameters do I have?

How to Deal with Large Input Spaces

- How can we apply neural nets to images?
- Images can have millions of pixels, i.e., $\mathbf{x}$ is very high dimensional

How to Deal with Large Input Spaces

- How can we apply neural nets to images?
- Images can have millions of pixels, i.e., $\mathbf{x}$ is very high dimensional
- How many parameters do I have?
- Prohibitive to have fully-connected layers
- What can we do?

How to Deal with Large Input Spaces

- How can we apply neural nets to images?
- Images can have millions of pixels, i.e., $\mathbf{x}$ is very high dimensional
- How many parameters do I have?
- Prohibitive to have fully-connected layers

How to Deal with Large Input Spaces

- How can we apply neural nets to images?
- Images can have millions of pixels, i.e., $\mathbf{x}$ is very high dimensional
- How many parameters do I have?
- Prohibitive to have fully-connected layers
- What can we do?
- We can use a locally connected layer

Locally Connected Layer

When Will this Work?

When Will this Work?

When Will this Work?

When Will this Work?

- This is good when the input is (roughly) registered

General Images

- The object can be anywhere

[Slide: Y. Zhu]

General Images

- The object can be anywhere

[Slide: Y. Zhu]

Convolutional Neural Net

- Idea: statistics are similar at different locations (Lecun 1998)
- Connect each hidden unit to a small input patch and share the weight across space
- This is called a convolution layer and the network is a convolutional network

General Images

- The object can be anywhere

[Slide: Y. Zhu]

Convolutional Layer

Ranzato

$$h_j^n = \max(0, \sum_{k=1}^{K} h_k^{n-1} * w_{jk}^n)$$

Convolutional Layer

Ranzato

$$h_j^n = \max(0, \sum_{k=1}^{K} h_k^{n-1} * w_{jk}^n)$$

Convolutional Layer

Ranzato

$$h_j^n = \max(0, \sum_{k=1}^{K} h_k^{n-1} * w_{jk}^n)$$

Convolutional Layer

Ranzato

$$h_j^n = \max(0, \sum_{k=1}^{K} h_k^{n-1} * w_{jk}^n)$$

Convolutional Layer

Ranzato

$$h_j^n = \max(0, \sum_{k=1}^{K} h_k^{n-1} * w_{jk}^n)$$

Convolutional Layer

Ranzato

$$h_j^n = \max(0, \sum_{k=1}^{K} h_k^{n-1} * w_{jk}^n)$$

Convolutional Layer

Learn multiple filters.

E.g.: 200x200 image
100 Filters
Filter size: 10x10
10K parameters

54

Ranzato

Now let's make this very deep to get a real state-of-the-art object recognition system

Convolutional Neural Networks (CNN)

- If our filter is $[-1, 1]$, you get a vertical edge detector

Convolutional Neural Networks (CNN)

- Remember a lecture on filtering?

Convolutional Neural Networks (CNN)

- So applying a filterbank to an image yields a cube-like output, a 3D matrix in which each slice is an output of convolution with one filter. We apply an activation function on each hidden unit (typically a ReLU).

Convolutional Neural Networks (CNN)

- Now imagine we want to have many filters (e.g., vertical, horizontal, corners, one for dots). We will use a filterbank.

Convolutional Layer

Figure: **Left:** CNN, **right:** Each neuron computes a linear and activation function

Convolutional Neural Networks (CNN)

- So applying a filterbank to an image yields a cube-like output, a 3D matrix in which each slice is an output of convolution with one filter. We apply an activation function on each hidden unit (typically a ReLU).

Convolutional Layer

Figure: **Left:** CNN, **right:** Each neuron computes a linear and activation function

Hyperparameters of a convolutional layer:

- The number of filters (controls the **depth** of the output volume)

Convolutional Layer

Figure: **Left:** CNN, **right:** Each neuron computes a linear and activation function

Hyperparameters of a convolutional layer:

Convolutional Layer

Figure: **Left:** CNN, **right:** Each neuron computes a linear and activation function

Hyperparameters of a convolutional layer:

- The number of filters (controls the **depth** of the output volume)
- The **stride**: how many units apart do we apply a filter spatially (this controls the spatial size of the output volume)
- The size $w \times h$ of the filters

[http://cs231n.github.io/convolutional-networks/]

Convolutional Layer

Figure: **Left:** CNN, **right:** Each neuron computes a linear and activation function

Hyperparameters of a convolutional layer:

- The number of filters (controls the **depth** of the output volume)
- The **stride**: how many units apart do we apply a filter spatially (this controls the spatial size of the output volume)

Convolutional Neural Networks (CNN)

- Do some additional tricks. A popular one is called max pooling. Any idea why you would do this?

MLP vs ConvNet

Figure: **Top:** MLP, **bottom:** Convolutional neural network

[http://cs231n.github.io/convolutional-networks/]

Pooling Layer

By "pooling" (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

61

Ranzato

Convolutional Neural Networks (CNN)

- Do some additional tricks. A popular one is called max pooling. Any idea why you would do this? To get invariance to small shifts in position.

$$O(i,j) = \max_{\substack{k \in \{i-1,i,i+1\} \\ l \in \{j-1,j,j+1\}}} O(k,l)$$

Take each slice in the output cube, and in each pixel compute a max over a small patch around it. This is called **max pooling**.

Pooling

Figure: **Left:** Pooling, **right:** max pooling example

Pooling Options

- Max Pooling: return the maximal argument
- Average Pooling: return the average of the arguments
- Other types of pooling exist.

Pooling

Figure: **Left:** Pooling, **right:** max pooling example

Hyperparameters of a pooling layer:

- The spatial extent F

Pooling

Figure: **Left:** Pooling, **right:** max pooling example

Hyperparameters of a pooling layer:

Pooling Layer: Receptive Field Size

If convolutional filters have size KxK and stride 1, and pooling layer has pools of size PxP, then each unit in the pooling layer depends upon a patch (at the input of the preceding conv. layer) of size: (P+K-1)x(P+K-1)

67

Ranzato

Pooling

Figure: **Left:** Pooling, **right:** max pooling example

Hyperparameters of a pooling layer:

- The spatial extent F
- The stride

[http://cs231n.github.io/convolutional-networks/]

Convolutional Neural Networks (CNN)

- Keep adding a few layers. Any idea what's the purpose of more layers? Why can't we just have a full bunch of filters in one layer?

Do it recursively

Have multiple ``layers''

Convolutional Neural Networks (CNN)

- Now add another "layer" of filters. For each filter again do convolution, but this time with the output cube of the previous layer.

Add one more layer of filters

These filters are convolved with the output of the previous layer. The results of each convolution is again a slice in the cube on the right.

What is the dimension of each of these filters?

Convolutional Neural Networks (CNN)

- Add one final layer: a classification layer. Each dimension of this vector tells us the probability of the input image being of a certain class.

Add a **classification** ``layer''.

For an input image, the value in a particular dimension of this vector tells you the probability of the corresponding object class.

Convolutional Neural Networks (CNN)

- In the end add one or two fully (or densely) connected layers. In this layer, we don't do convolution we just do a dot-product between the "filter" and the output of the previous layer.

In the top, most networks add a ``densely'' connected layer. You can think of this as a filter, and the output value is a dot product between the filter and the output cube of the previous layer.

What are the dimensions of this filter in this example? How many such filters are on this layer?

Convolutional Neural Networks (CNN)

- Or when the network sees a picture of a cat, the last layer will say "cat".

Convolutional Neural Networks (CNN)

- The trick is to not hand-fix the weights, but to train them. Train them such that when the network sees a picture of a dog, the last layer will say "dog".

Classification

- Once trained we feed in an image or a crop, run through the network, and read out the class with the highest probability in the last (classif) layer.

Convolutional Neural Networks (CNN)

- Or when the network sees a picture of a boat, the last layer will say "boat"... The more pictures the network sees, the better.

Train on **lots** of examples. Millions. Tens of millions. Wait a week for training to finish.

Share your network (the weights) with others who are not fortunate enough with GPU power.

Example

[http://cs231n.github.io/convolutional-networks/]

Architecture for Classification

Total nr. params: 60M

Total nr. flops: 832M

4M	LINEAR	4M
16M	FULLY CONNECTED	16M
37M	FULLY CONNECTED	37M
	MAX POOLING	
442K	CONV	74M
1.3M	CONV	224M
884K	CONV	149M
	MAX POOLING	
	LOCAL CONTRAST NORM	
307K	CONV	223M
	MAX POOLING	
	LOCAL CONTRAST NORM	
35K	CONV	105M

Krizhevsky et al. "ImageNet Classification with deep CNNs" NIPS 2012

Ranzato

Architecture for Classification

Krizhevsky et al. "ImageNet Classification with deep CNNs" NIPS 2012

Ranzato

ImageNet

- Imagenet, biggest dataset for object classification: `http://image-net.org/`
- 1000 classes, 1.2M training images, 150K for test

The 2012 Computer Vision Crisis

(Classification)

(Detection)

OpenImage

- OpenImage, the newest biggest dataset for object classification: `https://github.com/openimages/dataset`
- 6000 classes, 9 million

Open Images dataset

Open Images is a dataset of ~9 million URLs to images that have been annotated with labels spanning over 6000 categories.

The annotations are licensed by Google Inc. under CC BY 4.0 license. The contents of this repository are released under an Apache 2 license.

The images are listed as having a CC BY 2.0 license. **Note:** while we tried to identify images that are licensed under a Creative Commons Attribution license, we make no representations or warranties regarding the license status of each image and you should verify the license for each image yourself.

Goodies

- **New** Pretrained Inception v3 model is released.
- **New** OpenImages annotations on BigQuery
- Unofficial dataset viewer by tzatter.

So Neural Networks are Great

- But to train the networks you need quite a bit of computational power (e.g., GPU farm). So what do you do?

So Neural Networks are Great

- So networks turn out to be great.
- Everything is deep, even if it's shallow!
- Companies leading the competitions as they have more computational power
- At this point Google, Facebook, Microsoft, Baidu "steal" most neural network professors/students from academia

no need for social media

So Neural Networks are Great

- And train more layers. 16 instead of 7 before. 144 million parameters.

Figure: K. Simonyan, A. Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv 2014

So Neural Networks are Great

- Buy even more.

Results: Object Classification

Slide: R. Liao, Paper: [He, K., Zhang, X., Ren, S. and Sun, J., 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385, 2016]

150 Layers!

- Networks are now at 150 layers
- They use a skip connections with special form
- In fact, they don't fit on this screen
- Amazing performance!
- A lot of "mistakes" are due to wrong ground-truth

[He, K., Zhang, X., Ren, S. and Sun, J., 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385, 2016]

Results: Object Detection

Slide: R. Liao, Paper: [He, K., Zhang, X., Ren, S. and Sun, J., 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385, 2016]

Results: Object Detection

Slide: R. Liao, Paper: [He, K., Zhang, X., Ren, S. and Sun, J., 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385, 2016]

Results: Object Detection

Slide: R. Liao, Paper: [He, K., Zhang, X., Ren, S. and Sun, J., 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385, 2016]

Results: Object Detection

Slide: R. Liao, Paper: [He, K., Zhang, X., Ren, S. and Sun, J., 2015. Deep Residual Learning for Image Recognition.

What do CNNs Learn?

Figure: Filters in the second layer

[http://arxiv.org/pdf/1311.2901v3.pdf]

What do CNNs Learn?

Figure: Filters in the first convolutional layer of Krizhevsky et al

What do CNNs Learn?

[http://arxiv.org/pdf/1311.2901v3.pdf]

What do CNNs Learn?

Figure: Filters in the third layer

[http://arxiv.org/pdf/1311.2901v3.pdf]

How to Train Good CNNs

- Normalize your data (standard trick: subtract mean, divide by standard deviation)
- Augment your data (add image flips, rotations, etc)

How to Train Good CNNs

- Normalize your data (standard trick: subtract mean, divide by standard deviation)

How to Train Good CNNs

- Normalize your data (standard trick: subtract mean, divide by standard deviation)
- Augment your data (add image flips, rotations, etc)
- Keep training data balanced
- Shuffle data before batching

How to Train Good CNNs

- Normalize your data (standard trick: subtract mean, divide by standard deviation)
- Augment your data (add image flips, rotations, etc)
- Keep training data balanced

How to Train Good CNNs

- Normalize your data (standard trick: subtract mean, divide by standard deviation)
- Augment your data (add image flips, rotations, etc)
- Keep training data balanced
- Shuffle data before batching
- In training: Random initialization of weights with proper variance
- Monitor your loss function, and accuracy (performance) on validation

How to Train Good CNNs

- Normalize your data (standard trick: subtract mean, divide by standard deviation)
- Augment your data (add image flips, rotations, etc)
- Keep training data balanced
- Shuffle data before batching
- In training: Random initialization of weights with proper variance

How to Train Good CNNs

- Normalize your data (standard trick: subtract mean, divide by standard deviation)
- Augment your data (add image flips, rotations, etc)
- Keep training data balanced
- Shuffle data before batching
- In training: Random initialization of weights with proper variance
- Monitor your loss function, and accuracy (performance) on validation
- If your labeled image dataset is small: pre-train your CNN on a large dataset (eg Imagenet), and fine-tune on your dataset
- Sometime you may get "nan" values, this is quite likely due to e.g., $0 \cdot \log 0$. So don't forget to add an ϵ to your values

[Slide: Y. Zhu, check tutorial slides and code: http://www.cs.utoronto.ca/~fidler/teaching/2015/CSC2523.html]

How to Train Good CNNs

- Normalize your data (standard trick: subtract mean, divide by standard deviation)
- Augment your data (add image flips, rotations, etc)
- Keep training data balanced
- Shuffle data before batching
- In training: Random initialization of weights with proper variance
- Monitor your loss function, and accuracy (performance) on validation
- If your labeled image dataset is small: pre-train your CNN on a large dataset (eg Imagenet), and fine-tune on your dataset

More on NNs

Figure: Generate images: http://arxiv.org/pdf/1511.06434v1.pdf

Tricking a Neural Net

Read about it here (and try it!): https://codewords.recurse.com/issues/five/why-do-neural-networks-think-a-panda-is-a-vulture

Watch: https://www.youtube.com/watch?v=M2IebCN9Ht4

More on NNs

Generate text: https://vimeo.com/146492001, https://github.com/karpathy/neuraltalk2, https://github.com/ryankiros/visual-semantic-embedding

More on NNs

Figure: Compose music: https://www.youtube.com/watch?v=0VTI1BBLydE

Links

- NNs for computer vision: https://github.com/kjw0612/awesome-deep-vision
- Recurrent neural networks: https://github.com/kjw0612/awesome-rnn
- Lots of code, models, tutorials: https://github.com/carpedm20/awesome-torch
- More links on our class webpage