Recall

We have:

\[
\begin{align*}
    a_1^{(2)} &= f(W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 + W_{13}^{(1)} x_3 + b_{1}^{(1)}) \\
    a_2^{(2)} &= f(W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2 + W_{23}^{(1)} x_3 + b_{2}^{(1)}) \\
    a_3^{(2)} &= f(W_{31}^{(1)} x_1 + W_{32}^{(1)} x_2 + W_{33}^{(1)} x_3 + b_{3}^{(1)}) \\
    h_{W,b} &= a_1^{(3)} = f(W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + W_{13}^{(2)} a_3^{(2)} + b_1^{(2)}) .
\end{align*}
\]
Vector form:

\[
\begin{align*}
    z^{(2)} &= W^{(1)} x + b^{(1)} \\
    a^{(2)} &= f(z^{(2)}) \\
    z^{(3)} &= W^{(2)} a^{(2)} + b^{(2)} \\
    h_{W,b} &= a^{(3)} = f(z^{(3)}).
\end{align*}
\]
Suppose we have

- A neural network with $s_l$ neurons in layer $l$ ($l = 1, \ldots, n_l$).
Training neural networks

Suppose we have

- A neural network with $s_l$ neurons in layer $l$ ($l = 1, \ldots, n_l$).
- Observations $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)}) \in \mathbb{R}^{s_1} \times \mathbb{R}^{s_{n_l}}$.

We would like to choose $W^{(l)}$ and $b^{(l)}$ in some optimal way for all $l$. 

Let $J(W, b) := \frac{1}{2} \| h_{W,b}(x) - y \|_2^2$ (Squared error for one sample).

Define $J(W, b) := \frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) + \frac{\lambda}{2} \sum_{l=1}^{n_l} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l}+1} (W^{(l)}_{ij})^2.$ (average squared error with Ridge penalty).

Note: The Ridge penalty prevents overfitting.

We do not penalize the bias terms $b^{(l)}$. 

4/14
Suppose we have

- A neural network with \( s_l \) neurons in layer \( l \) (\( l = 1, \ldots, n_l \)).
- Observations \( (x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)}) \in \mathbb{R}^{s_1} \times \mathbb{R}^{s_{n_l}} \).

We would like to choose \( W^{(l)} \) and \( b^{(l)} \) in some optimal way for all \( l \).

Let

\[
J(W, b; x, y) := \frac{1}{2} \| h_{W, b}(x) - y \|_2^2 \quad \text{(Squared error for one sample)}.
\]
Training neural networks

Suppose we have

- A neural network with $s_l$ neurons in layer $l$ ($l = 1, \ldots, n_l$).
- Observations $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)}) \in \mathbb{R}^{s_1} \times \mathbb{R}^{s_{n_l}}$.

We would like to choose $W^{(l)}$ and $b^{(l)}$ in some optimal way for all $l$.

Let

$$J(W, b; x, y) := \frac{1}{2} \| h_{W, b}(x) - y \|_2^2$$

(Squared error for one sample).

Define

$$J(W, b) := \frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (W_{ji}^{(l)})^2.$$

(average squared error with Ridge penalty).
Suppose we have

- A neural network with $s_l$ neurons in layer $l$ ($l = 1, \ldots, n_l$).
- Observations $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)}) \in \mathbb{R}^{s_1} \times \mathbb{R}^{s_{n_l}}$.

We would like to choose $W^{(l)}$ and $b^{(l)}$ in some optimal way for all $l$.

Let

$$J(W, b; x, y) := \frac{1}{2} \| h_{W,b}(x) - y \|_2^2$$

(Squared error for one sample).

Define

$$J(W, b) := \frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (W_{ji}^{(l)})^2.$$  

(average squared error with Ridge penalty).

Note:

- The Ridge penalty prevents overfitting.
- We do not penalize the bias terms $b_i^{(l)}$. 

Some remarks

- Can use other loss functions (e.g. for classification).
- Can use other penalties (e.g. $\ell_1$, elastic net, etc.).
• Can use other loss functions (e.g. for classification).
• Can use other penalties (e.g. $\ell_1$, elastic net, etc.).
• In classification problems, we choose the labels $y \in \{0, 1\}$ (if working with sigmoid) or $y \in \{-1, 1\}$ (if working with tanh).
• For regression problems, we scale the output so that $y \in [0, 1]$ (if working with sigmoid) or $y \in [-1, 1]$ (if working with tanh).
Some remarks

- Can use other loss functions (e.g. for classification).
- Can use other penalties (e.g. $\ell_1$, elastic net, etc.).
- In classification problems, we choose the labels $y \in \{0, 1\}$ (if working with sigmoid) or $y \in \{-1, 1\}$ (if working with $\tanh$).
- For regression problems, we scale the output so that $y \in [0, 1]$ (if working with sigmoid) or $y \in [-1, 1]$ (if working with $\tanh$).
- We can use gradient descent to minimize $J(W, b)$. Note that since the function $J(W, b)$ is non-convex, we may only find a local minimum.
Some remarks

- Can use other loss functions (e.g. for classification).
- Can use other penalties (e.g. \( \ell_1 \), elastic net, etc.).
- In classification problems, we choose the labels \( y \in \{0, 1\} \) (if working with sigmoid) or \( y \in \{-1, 1\} \) (if working with \( \tanh \)).
- For regression problems, we scale the output so that \( y \in [0, 1] \) (if working with sigmoid) or \( y \in [-1, 1] \) (if working with \( \tanh \)).
- We can use gradient descent to minimize \( J(W, b) \). Note that since the function \( J(W, b) \) is non-convex, we may only find a local minimum.
- We need an initial choice for \( W_{ij}^{(l)} \) and \( b_i^{(l)} \). If we initialize all the parameters to 0, then the parameters remain constant over the layers because of the symmetry of the problem.
Some remarks

- Can use other loss functions (e.g. for classification).
- Can use other penalties (e.g. $\ell_1$, elastic net, etc.).
- In classification problems, we choose the labels $y \in \{0, 1\}$ (if working with sigmoid) or $y \in \{-1, 1\}$ (if working with tanh).
- For regression problems, we scale the output so that $y \in [0, 1]$ (if working with sigmoid) or $y \in [-1, 1]$ (if working with tanh).
- We can use gradient descent to minimize $J(W, b)$. Note that since the function $J(W, b)$ is non-convex, we may only find a local minimum.
- We need an initial choice for $W_{ij}^{(l)}$ and $b_i^{(l)}$. If we initialize all the parameters to 0, then the parameters remain constant over the layers because of the symmetry of the problem.
- As a result, we initialize the parameters to a small constant at random (say, using $N(0, \epsilon^2)$ for $\epsilon = 0.01$).
We update the parameters using a gradient descent as follows:

\[ W_{ij}^{(l)} \leftarrow W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) \]

\[ b_i^{(l)} \leftarrow b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b). \]

Here \( \alpha > 0 \) is a parameter (the learning rate).
Gradient descent and the backpropagation algorithm

- We update the parameters using a gradient descent as follows:

\[
W_{ij}^{(l)} \leftarrow W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b)
\]

\[
b_{i}^{(l)} \leftarrow b_{i}^{(l)} - \alpha \frac{\partial}{\partial b_{i}^{(l)}} J(W, b).
\]

Here \( \alpha > 0 \) is a parameter (the learning rate).

- The partial derivatives can be cleverly computed using the chain rule to avoid repeating calculations (backpropagation algorithm).
Sparse neural networks

Sparse networks can be built by

- Penalizing coefficients (e.g. using a $\ell_1$ penalty).
- Dropping some of the connections at random (dropout).

Srivastava et al., JMLR 15 (2014).

Useful to prevent overfitting.
Recent work: “One-shot learners” can be used to train models with a smaller sample size.
Autoencoders

An **autoencoder** learns the identity function:

- **Input**: unlabeled data.
- **Output** = input.
- **Idea**: limit the number of hidden layers to discover structure in the data.
- **Learn a *compressed* representation of the input.**

Source: UFLDL tutorial.
Train an autoencoder on $10 \times 10$ images with one hidden layer.
Train an autoencoder on $10 \times 10$ images with one hidden layer. Each hidden unit $i$ computes:

$$a_i^{(2)} = f \left( \sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_j^{(1)} \right).$$
Train an autoencoder on $10 \times 10$ images with one hidden layer.

Each hidden unit $i$ computes:

$$a_i^{(2)} = f \left( \sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_j^{(1)} \right).$$

Think of $a_i^{(2)}$ as some non-linear feature of the input $x$. 

Problem: Find $x$ that maximally activates $a_i^{(2)}$ over $\|x\|_2 \leq 1$.

Claim: $x_j = W_{ij}^{(1)} \sqrt{\sum_{j=1}^{100} (W_{ij}^{(1)})^2}$. (Hint: Use Cauchy-Schwarz.)

We can now display the image maximizing $a_i^{(2)}$ for each $i$. 

Train an autoencoder on $10 \times 10$ images with one hidden layer.

Each hidden unit $i$ computes:

$$a^{(2)}_i = f \left( \sum_{j=1}^{100} W^{(1)}_{ij} x_j + b^{(1)}_j \right).$$

Think of $a^{(2)}_i$ as some non-linear feature of the input $x$.

**Problem:** Find $x$ that maximally activates $a^{(2)}_i$ over $\|x\|_2 \leq 1$. 

(Hint: Use Cauchy-Schwartz.)
Train an autoencoder on $10 \times 10$ images with one hidden layer.

Each hidden unit $i$ computes:

$$a_i^{(2)} = f \left( \sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_j^{(1)} \right).$$

Think of $a_i^{(2)}$ as some non-linear feature of the input $x$.

**Problem:** Find $x$ that maximally activates $a_i^{(2)}$ over $\|x\|_2 \leq 1$.

**Claim:**

$$x_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{100} (W_{ij}^{(1)})^2}}.$$
• Train an autoencoder on $10 \times 10$ images with one hidden layer.
• Each hidden unit $i$ computes:

$$a_i^{(2)} = f \left( \sum_{j=1}^{100} W_{ij} x_j + b_j^{(1)} \right).$$

• Think of $a_i^{(2)}$ as some non-linear feature of the input $x$.

**Problem:** Find $x$ that maximally activates $a_i^{(2)}$ over $\|x\|_2 \leq 1$.

**Claim:**

$$x_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{100} (W_{ij}^{(1)})^2}}.$$

(Hint: Use Cauchy–Schwarz).

We can now display the image maximizing $a_i^{(2)}$ for each $i$. 
100 hidden units on 10x10 pixel inputs:

The different hidden units have learned to detect edges at different positions and orientations in the image.
Idea: Certain signals are *stationary*, i.e., their statistical properties do not change in space or time.
Using convolutions

- Idea: Certain signals are stationary, i.e., their statistical properties do not change in space or time.
- For example, images often have similar statistical properties in different regions in space.
Using convolutions

- Idea: Certain signals are *stationary*, i.e., their statistical properties do not change in space or time.
- For example, images often have similar statistical properties in different regions in space.
- That suggests that the features that we learn at one part of an image can also be applied to other parts of the image.

Source: UFLDL tutorial.
Using convolutions

- Idea: Certain signals are *stationary*, i.e., their statistical properties do not change in space or time.
- For example, images often have similar statistical properties in different regions in space.
- That suggests that the features that we learn at one part of an image can also be applied to other parts of the image.
- Can “convolve” the learned features with the larger image.
Idea: Certain signals are *stationary*, i.e., their statistical properties do not change in space or time.

For example, images often have similar statistical properties in different regions in space.

That suggests that the features that we learn at one part of an image can also be applied to other parts of the image.

Can “convolve” the learned features with the larger image.

**Example:** $96 \times 96$ image.
Using convolutions

- Idea: Certain signals are *stationary*, i.e., their statistical properties do not change in space or time.
- For example, images often have similar statistical properties in different regions in space.
- That suggests that the features that we learn at one part of an image can also be applied to other parts of the image.
- Can “convolve” the learned features with the larger image.

**Example:** 96 × 96 image.
- Learn features on small 8 × 8 patches sampled randomly (e.g. using a sparse autoencoder).
Using convolutions

- Idea: Certain signals are stationary, i.e., their statistical properties do not change in space or time.
- For example, images often have similar statistical properties in different regions in space.
- That suggests that the features that we learn at one part of an image can also be applied to other parts of the image.
- Can “convolve” the learned features with the larger image.

Example: 96 × 96 image.
- Learn features on small 8 × 8 patches sampled randomly (e.g. using a sparse autoencoder).
- Run the trained model through all 8 × 8 patches of the image to get the feature activations.
Using convolutions

- Idea: Certain signals are *stationary*, i.e., their statistical properties do not change in space or time.
- For example, images often have similar statistical properties in different regions in space.
- That suggests that the features that we learn at one part of an image can also be applied to other parts of the image.
- Can “convolve” the learned features with the larger image.

**Example:** $96 \times 96$ image.

- Learn features on small $8 \times 8$ patches sampled randomly (e.g. using a sparse autoencoder).
- Run the trained model through all $8 \times 8$ patches of the image to get the feature activations.

Source: UFLDL tutorial.
Pooling features

- Once can also *pool* the features obtained via convolution.

For example, to describe a large image, one natural approach is to aggregate statistics of these features at various locations. E.g. compute the mean, max, etc. over different regions. Can lead to more robust features. Can lead to invariant features. For example, if the pooling regions are contiguous, then the pooling units will be translation invariant, i.e., they won't change much if objects in the image are undergo a (small) translation.
Once can also *pool* the features obtained via convolution.

For example, to describe a large image, one natural approach is to aggregate statistics of these features at various locations.
Pooling features

- Once can also *pool* the features obtained via convolution.
- For example, to describe a large image, one natural approach is to aggregate statistics of these features at various locations.
- E.g. compute the mean, max, etc. over different regions.
Pooling features

- Once can also *pool* the features obtained via convolution.
- For example, to describe a large image, one natural approach is to aggregate statistics of these features at various locations.
- E.g. compute the mean, max, etc. over different regions.
- Can lead to more robust features. Can lead to invariant features.
Pooling features

- Once can also pool the features obtained via convolution.
- For example, to describe a large image, one natural approach is to aggregate statistics of these features at various locations.
- E.g. compute the mean, max, etc. over different regions.
- Can lead to more robust features. Can lead to invariant features.
- For example, if the pooling regions are contiguous, then the pooling units will be “translation invariant”, i.e., they won’t change much if objects in the image are undergo a (small) translation.
Pooling features

- Once can also *pool* the features obtained via convolution.
- For example, to describe a large image, one natural approach is to aggregate statistics of these features at various locations.
- E.g. compute the mean, max, etc. over different regions.
- Can lead to more robust features. Can lead to invariant features.
- For example, if the pooling regions are contiguous, then the pooling units will be “translation invariant”, i.e., they won’t change much if objects in the image are undergo a (small) translation.

![Diagram showing convolution and pooling of features](image-url)
We will use the package `h2o` to train neural networks with R. To get you started, we will construct a neural network with 1 hidden layer containing 2 neurons to learn the XOR function:

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

```r
# Initialize h2o
library(h2o)

h2o.init(nthreads=-1, max_mem_size="2G")
h2o.removeAll() # in case the cluster was already running

# Construct the XOR function
X = t(matrix(c(0,0,0,1,1,0,1,1),2,4))
y = matrix(c(-1,1,1,-1), 4)
train = as.h2o(cbind(X,y))
```
Training the model:

```r
# Train model
model <- h2o.deeplearning(x = names(train)[1:2],
                           y = names(train)[3],
                           training_frame = train,
                           activation = "Tanh",
                           hidden = c(2),
                           input_dropout_ratio = 0.0,
                           l1 = 0,
                           epochs = 10000)

# Test the model
h2o.predict(model, train)
```

Some options you may want to use when building more complicated models for data:

```r
activation = "RectifierWithDropout"
input_dropout_ratio = 0.2
l1 = 1e-5
```