Foundation of Machine Learning

Amir H. Payberah
payberah@kth.se
2020-09-28
The Course Web Page

https://fid3024.github.io
Linear Regression
Given the dataset of \( m \) houses.

<table>
<thead>
<tr>
<th>Living area</th>
<th>No. of bedrooms</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2104</td>
<td>3</td>
<td>400</td>
</tr>
<tr>
<td>1600</td>
<td>3</td>
<td>330</td>
</tr>
<tr>
<td>2400</td>
<td>3</td>
<td>369</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Predict the prices of other houses, as a function of the size of living area and number of bedrooms?
Building a model that takes input $\mathbf{x} \in \mathbb{R}^n$ and predicts output $\hat{y} \in \mathbb{R}$. 
Building a model that takes input $x \in \mathbb{R}^n$ and predicts output $\hat{y} \in \mathbb{R}$.

In linear regression, the output $\hat{y}$ is a linear function of the input $x$.

$$\hat{y} = f_w(x) = w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

$$\hat{y} = w^\top x$$

- $\hat{y}$: the predicted value
- $n$: the number of features
- $x_i$: the $i$th feature value
- $w_j$: the $j$th model parameter ($w \in \mathbb{R}^n$)
Loss Function

- For each value of the $w$, how close the $\hat{y}^{(i)}$ is to the corresponding $y^{(i)}$.
- E.g., Mean Squared Error (MSE)

$$J(w) = \frac{1}{m} \sum_{i=1}^{m} \text{cost}_w(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)})^2$$
Objective

- Minimizing the loss function $J(w)$.
- Gradient descent
Gradient Descent

- Tweaking parameters \( w \) iteratively in order to minimize a loss function \( J(w) \).
Gradient Descent

- **Tweaking parameters** $\mathbf{w}$ **iteratively** in order to **minimize** a loss function $J(\mathbf{w})$.

- Start at a **random point**, and repeat the following **steps**, until the **stopping criterion** is satisfied:

  1. Determine a descent direction $\nabla J(\mathbf{w})$
  2. Choose a step size $\eta$
  3. Update the parameters:
     $$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla J(\mathbf{w})$$
Gradient Descent

- Tweaking parameters $w$ iteratively in order to minimize a loss function $J(w)$.

- Start at a random point, and repeat the following steps, until the stopping criterion is satisfied:
  1. Determine a descent direction $\nabla J(w)$

![Diagram of Gradient Descent](image)
Gradient Descent

- Tweaking parameters $w$ iteratively in order to minimize a loss function $J(w)$.

- Start at a random point, and repeat the following steps, until the stopping criterion is satisfied:
  1. Determine a descent direction $\nabla J(w)$
  2. Choose a step size $\eta$
Gradient Descent

- Tweaking parameters $w$ iteratively in order to minimize a loss function $J(w)$.

- Start at a random point, and repeat the following steps, until the stopping criterion is satisfied:
  1. Determine a descent direction $\nabla J(w)$
  2. Choose a step size $\eta$
  3. Update the parameters: $w \leftarrow w - \eta \nabla J(w)$
Batch Gradient Descent vs. Mini-Batch Stochastic Gradient Descent

- **Gradient descent**
  - $\mathbf{X}$ is the total dataset.
  - $J(\mathbf{w}) = \frac{1}{|\mathbf{X}|} \sum_{\mathbf{x} \in \mathbf{X}} \text{cost}_w(y^{(i)}, \hat{y}^{(i)})$
Batch Gradient Descent vs. Mini-Batch Stochastic Gradient Descent

- **Gradient descent**
  - $\mathbf{X}$ is the **total dataset**.
  - $J(\mathbf{w}) = \frac{1}{|\mathbf{X}|} \sum_{\mathbf{x} \in \mathbf{X}} \text{cost}_w(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{|\mathbf{X}|} \sum_{\mathbf{x} \in \mathbf{X}} \ell(\mathbf{x}, \mathbf{w})$

- **Mini-batch stochastic gradient descent**
  - $\beta$ is the mini-batch, i.e., a random subset of $\mathbf{X}$.
  - $J(\mathbf{w}) = \frac{1}{|\beta|} \sum_{\mathbf{x} \in \beta} \text{cost}_w(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{|\beta|} \sum_{\mathbf{x} \in \beta} \ell(\mathbf{x}, \mathbf{w})$
Batch Gradient Descent vs. Mini-Batch Stochastic Gradient Descent

- **Gradient descent**
  - $X$ is the total dataset.
  - $J(w) = \frac{1}{|X|} \sum_{x \in X} \text{cost}_w(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{|X|} \sum_{x \in X} l(x, w)$
  - $w \leftarrow w - \eta \frac{1}{|X|} \sum_{x \in X} \nabla l(x, w)$
Batch Gradient Descent vs. Mini-Batch Stochastic Gradient Descent

- **Gradient descent**
  - \( X \) is the total dataset.
  - \( J(w) = \frac{1}{|X|} \sum_{x \in X} \text{cost}_w(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{|X|} \sum_{x \in X} l(x, w) \)
  - \( w \leftarrow w - \eta \frac{1}{|X|} \sum_{x \in X} \nabla l(x, w) \)

- **Mini-batch stochastic gradient descent**
  - \( \beta \) is the mini-batch, i.e., a random subset of \( X \).
  - \( J(w) = \frac{1}{|\beta|} \sum_{x \in \beta} \text{cost}_w(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{|\beta|} \sum_{x \in \beta} l(x, w) \)
Batch Gradient Descent vs. Mini-Batch Stochastic Gradient Descent

► Gradient descent
  • $X$ is the total dataset.
  • $J(w) = \frac{1}{|X|} \sum_{x \in X} cost_w(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{|X|} \sum_{x \in X} l(x, w)$
  • $w \leftarrow w - \eta \frac{1}{|X|} \sum_{x \in X} \nabla l(x, w)$

► Mini-batch stochastic gradient descent
  • $\beta$ is the mini-batch, i.e., a random subset of $X$.
  • $J(w) = \frac{1}{|\beta|} \sum_{x \in \beta} cost_w(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{|\beta|} \sum_{x \in \beta} l(x, w)$
  • $w \leftarrow w - \eta \frac{1}{|\beta|} \sum_{x \in \beta} \nabla l(x, w)$
Binomial Logistic Regression
Binomial Logistic Regression (1/2)

Given the dataset of \( m \) cancer tests.

<table>
<thead>
<tr>
<th>Tumor size</th>
<th>Cancer</th>
</tr>
</thead>
<tbody>
<tr>
<td>330</td>
<td>1</td>
</tr>
<tr>
<td>120</td>
<td>0</td>
</tr>
<tr>
<td>400</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Predict the risk of cancer, as a function of the tumor size?
Binomial Logistic Regression (2/2)

- **Linear regression**: the model computes the *weighted sum of the input features* (plus a bias term).

\[
\hat{y} = w_0 x_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{w}^\top \mathbf{x}
\]
**Linear regression**: the model computes the **weighted sum of the input features** (plus a bias term).

\[
\hat{y} = w_0x_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n = \mathbf{w}^\top \mathbf{x}
\]

**Binomial logistic regression**: the model computes a **weighted sum of the input features** (plus a bias term), but it **outputs the logistic of this result**.

\[
z = w_0x_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n = \mathbf{w}^\top \mathbf{x}
\]

\[
\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}}}
\]
Naive idea: minimizing the Mean Squared Error (MSE)

\[ \text{cost}(\hat{y}^{(i)}, y^{(i)}) = (\hat{y}^{(i)} - y^{(i)})^2 \]

\[ J(w) = \frac{1}{m} \sum_{i} \text{cost}(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{m} \sum_{i} (\hat{y}^{(i)} - y^{(i)})^2 \]
Naive idea: minimizing the Mean Squared Error (MSE)

\[
\text{cost}(\hat{y}^{(i)}, y^{(i)}) = (\hat{y}^{(i)} - y^{(i)})^2
\]

\[
J(w) = \frac{1}{m} \sum_{i} \text{cost}(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{m} \sum_{i} (\hat{y}^{(i)} - y^{(i)})^2
\]

\[
J(w) = \text{MSE}(w) = \frac{1}{m} \sum_{i} \left( \frac{1}{1 + e^{-w^T x^{(i)}}} - y^{(i)} \right)^2
\]
Loss Function (1/3)

- **Naive idea:** minimizing the Mean Squared Error (MSE)

\[
\text{cost}(\hat{y}^{(i)}, y^{(i)}) = (\hat{y}^{(i)} - y^{(i)})^2
\]

\[
J(w) = \frac{1}{m} \sum_{i} \text{cost}(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{m} \sum_{i} (\hat{y}^{(i)} - y^{(i)})^2
\]

\[
J(w) = \text{MSE}(w) = \frac{1}{m} \sum_{i} \left( \frac{1}{1 + e^{-w^T x^{(i)}}} - y^{(i)} \right)^2
\]

- This cost function is a non-convex function for parameter optimization.
Loss Function (2/3)

\[
\text{cost}(\hat{y}^{(i)}, y^{(i)}) = \begin{cases} 
  -\log(\hat{y}^{(i)}) & \text{if } y^{(i)} = 1 \\
  -\log(1 - \hat{y}^{(i)}) & \text{if } y^{(i)} = 0
\end{cases}
\]
We can define $J(w)$ as below

$$
cost(\hat{y}^{(i)}, y^{(i)}) = \begin{cases} 
-\log(\hat{y}^{(i)}) & \text{if } y^{(i)} = 1 \\
-\log(1 - \hat{y}^{(i)}) & \text{if } y^{(i)} = 0 
\end{cases}
$$
We can define $J(w)$ as below

$$\text{cost}(\hat{y}^{(i)}, y^{(i)}) = \begin{cases} 
-\log(\hat{y}^{(i)}) & \text{if } y^{(i)} = 1 \\
-\log(1 - \hat{y}^{(i)}) & \text{if } y^{(i)} = 0 
\end{cases}$$

$$J(w) = \frac{1}{m} \sum_{i} \text{cost}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i} (y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}))$$
In a binomial classifier, $y \in \{0, 1\}$, the estimator is $\hat{y} = p(y = 1 \mid x; w)$.

- We find one set of parameters $w$.

$$w^T = [w_0, w_1, \cdots, w_n]$$
Binomial vs. Multinomial Logistic Regression (1/2)

- In a binomial classifier, \( y \in \{0, 1\} \), the estimator is \( \hat{y} = p(y = 1 | x; w) \).
  - We find one set of parameters \( w \).

  \[
  w^T = [w_0, w_1, \ldots, w_n]
  \]

- In multinomial classifier, \( y \in \{1, 2, \ldots, k\} \), we need to estimate the result for each individual label, i.e., \( \hat{y}_j = p(y = j | x; w) \).
Binomial vs. Multinomial Logistic Regression (1/2)

- In a **binomial classifier**, \( y \in \{0, 1\} \), the **estimator** is \( \hat{y} = p(y = 1 \mid x; w) \).
  - We find **one** set of parameters \( w \).

\[
\mathbf{w}^T = [w_0, w_1, \cdots, w_n]
\]

- In **multinomial classifier**, \( y \in \{1, 2, \cdots, k\} \), we need to estimate the result for each individual label, i.e., \( \hat{y}_j = p(y = j \mid x; w) \).
  - We find **k** set of parameters \( \mathbf{W} \).

\[
\mathbf{W} = \begin{bmatrix}
[w_{0,1}, w_{1,1}, \cdots, w_{n,1}] \\
[w_{0,2}, w_{1,2}, \cdots, w_{n,2}] \\
\vdots \\
[w_{0,k}, w_{1,k}, \cdots, w_{n,k}]
\end{bmatrix} = \begin{bmatrix}
\mathbf{w}_1^T \\
\mathbf{w}_2^T \\
\vdots \\
\mathbf{w}_k^T
\end{bmatrix}
\]
In a binary class, $y \in \{0, 1\}$, we use the sigmoid function.

$$\mathbf{w}^T \mathbf{x} = w_0 x_0 + w_1 x_1 + \cdots + w_n x_n$$

$$\hat{y} = p(y = 1 \mid \mathbf{x}; \mathbf{w}) = \sigma(\mathbf{w}^T \mathbf{x}) = \frac{1}{1 + e^{-w^T \mathbf{x}}}$$
Binomial vs. Multinomial Logistic Regression (2/2)

- In a binary class, $y \in \{0, 1\}$, we use the sigmoid function.

$$\mathbf{w}^T \mathbf{x} = w_0 x_0 + w_1 x_1 + \cdots + w_n x_n$$

$$\hat{y} = p(y = 1 \mid \mathbf{x}; \mathbf{w}) = \sigma(\mathbf{w}^T \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}}$$

- In multiclass, $y \in \{1, 2, \cdots, k\}$, we use the softmax function.

$$\mathbf{w}_j^T \mathbf{x} = w_{0,j} x_0 + w_{1,j} x_1 + \cdots + w_{n,j} x_n, \quad 1 \leq j \leq k$$

$$\hat{y}_j = p(y = j \mid \mathbf{x}; \mathbf{w}_j) = \sigma(\mathbf{w}_j^T \mathbf{x}) = \frac{e^{\mathbf{w}_j^T \mathbf{x}}}{\sum_{i=1}^{k} e^{\mathbf{w}_i^T \mathbf{x}}}$$
Sigmoid vs. Softmax

- **Sigmoid** function: \( \sigma(w^T x) = \frac{1}{1 + e^{-w^T x}} \)

- **Softmax** function: \( \sigma(w_j^T x) = \frac{e^{w_j^T x}}{\sum_{i=1}^{k} e^{w_i^T x}} \)
  - Calculate the probabilities of each target class over all possible target classes.
  - The softmax function for **two classes** is equivalent to the sigmoid function.
Deep Neural Network
The Linear Threshold Unit (LTU)

- Each input connection is associated with a weight.
- Computes a weighted sum of its inputs and applies a step function to that sum.

\[ z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = w^T x \]

\[ \hat{y} = \text{step}(z) = \text{step}(w^T x) \]
The Perceptron

- The perceptron is a single layer of LTUs.
- Train the model.
The perceptron is a single layer of LTUs.

Train the model.

\[ \hat{y} = f_w(X) \]
\[ J(w) = \text{cost}(y, \hat{y}) \]
\[ w \leftarrow w - \eta \nabla J(w) \]
A feedforward neural network is composed of:

- One input layer
- One or more hidden layers
- One final output layer
How to train a feedforward neural network?

1. Forward pass: make a prediction (i.e., $\hat{y}(i)$).
2. Measure the error (i.e., $\text{cost}(\hat{y}(i), y(i))$).
3. Backward pass: go through each layer in reverse to measure the error contribution from each connection.
4. Tweak the connection weights to reduce the error (update $W$ and $b$).
How to train a feedforward neural network?

For each training instance $x^{(i)}$ the algorithm does the following steps:

1. Forward pass: make a prediction (i.e., $\hat{y}^{(i)}$).
2. Measure the error (i.e., $cost(\hat{y}^{(i)}, y^{(i)})$).
3. Backward pass: go through each layer in reverse to measure the error contribution from each connection.
4. Tweak the connection weights to reduce the error (update $W$ and $b$).

It's called the backpropagation training algorithm.
How to train a feedforward neural network?

For each training instance $\mathbf{x}^{(i)}$ the algorithm does the following steps:

1. **Forward pass**: make a prediction (i.e., $\hat{y}^{(i)}$).
How to train a feedforward neural network?

For each training instance $x^{(i)}$ the algorithm does the following steps:

1. **Forward pass**: make a prediction (i.e., $\hat{y}^{(i)}$).
2. Measure the error (i.e., $\text{cost}(\hat{y}^{(i)}, y^{(i)})$).
How to train a feedforward neural network?

For each training instance \( x^{(i)} \) the algorithm does the following steps:

1. **Forward pass**: make a prediction (i.e., \( \hat{y}^{(i)} \)).
2. Measure the error (i.e., \( \text{cost}(\hat{y}^{(i)}, y^{(i)}) \)).
3. **Backward pass**: go through each layer in reverse to measure the error contribution from each connection.
How to train a *feedforward neural network*?

For each training instance $x^{(i)}$ the algorithm does the following steps:

1. **Forward pass**: make a prediction (i.e., $\hat{y}^{(i)}$).
2. Measure the error (i.e., $\text{cost}(\hat{y}^{(i)}, y^{(i)})$).
3. **Backward pass**: go through each layer in reverse to measure the error contribution from each connection.
4. Tweak the connection weights to reduce the error (update $W$ and $b$).
How to train a feedforward neural network?

- For each training instance $\mathbf{x}^{(i)}$ the algorithm does the following steps:
  1. **Forward pass**: make a prediction (i.e., $\mathbf{\hat{y}}^{(i)}$).
  2. Measure the **error** (i.e., $\text{cost}(\mathbf{\hat{y}}^{(i)}, y^{(i)})$).
  3. **Backward pass**: go through each layer in reverse to measure the **error contribution** from each connection.
  4. **Tweak the connection weights** to **reduce the error** (update $\mathbf{W}$ and $\mathbf{b}$).

- It’s called the **backpropagation** training algorithm.
Generalization
Generalization

- **Generalization**: make a model that performs well on test data.
  - Have a small test error.
Generalization

- **Generalization**: make a model that performs well on test data.
  - Have a small test error.
- **Challenges**
  1. Make the training error small.
  2. Make the gap between training and test error small.
Generalization

- Generalization: make a model that performs well on test data.
  - Have a small test error.

- Challenges
  1. Make the training error small.
  2. Make the gap between training and test error small.

- Overfitting vs. underfitting

[https://ml.berkeley.edu/blog/2017/07/13/tutorial-4]
Avoiding Overfitting

- Early stopping
- $l_1$ and $l_2$ regularization
- Max-norm regularization
- Dropout
- Data augmentation
Early Stopping

As the training steps go by, its prediction error on the training/validation set naturally goes down.
Early Stopping

▶ As the training steps go by, its prediction error on the training/validation set naturally goes down.

▶ After a while the validation error stops decreasing and starts to go back up.
  • The model has started to overfit the training data.
Early Stopping

- As the training steps go by, its prediction error on the training/validation set naturally goes down.

- After a while the validation error stops decreasing and starts to go back up.
  - The model has started to overfit the training data.

- In the early stopping, we stop training when the validation error reaches a minimum.

![Learning curves graph](image)
\textbf{L1 and L2 Regularization}

- Penalize \textbf{large values} of weights $w_j$.

\[
\tilde{J}(w) = J(w) + \lambda R(w)
\]
\section*{\(l_1\) and \(l_2\) Regularization}

- **Penalize large values** of weights \(w_j\).

\[ \tilde{J}(w) = J(w) + \lambda R(w) \]

- **\(l_1\) regression**: \(R(w) = \lambda \sum_{i=1}^{n} |w_i|\) is added to the cost function.

\[ \tilde{J}(w) = J(w) + \lambda \sum_{i=1}^{n} |w_i| \]
/1 and /2 Regularization

- Penalize large values of weights $w_j$.

$$\tilde{J}(w) = J(w) + \lambda R(w)$$

- /1 regression: $R(w) = \lambda \sum_{i=1}^{n} |w_i|$ is added to the cost function.

$$\tilde{J}(w) = J(w) + \lambda \sum_{i=1}^{n} |w_i|$$

- /2 regression: $R(w) = \lambda \sum_{i=1}^{n} w_i^2$ is added to the cost function.

$$\tilde{J}(w) = J(w) + \lambda \sum_{i=1}^{n} w_i^2$$
Max-norm regularization: constrains the weights $w_j$ of the incoming connections for each neuron $j$.

- Prevents them from getting too large.
Max-Norm Regularization

- **Max-norm regularization**: constrains the weights $w_j$ of the incoming connections for each neuron $j$.
  - Prevents them from getting too large.

- After each training step, clip $w_j$ as below, if $\|w_j\|_2 > r$:
  $$w_j \leftarrow \frac{r}{\|w_j\|_2} w_j$$
  - $r$ is the max-norm hyperparameter
  - $\|w_j\|_2 = (\sum_i w_{i,j}^2)^{\frac{1}{2}} = \sqrt{w_{1,j}^2 + w_{2,j}^2 + \cdots + w_{n,j}^2}$
At each training step, each neuron drops out temporarily with a probability $p$. The hyperparameter $p$ is called the dropout rate. A neuron will be entirely ignored during this training step. It may be active during the next step. Exclude the output neurons. After training, neurons don’t get dropped anymore.
At each training step, each neuron drops out temporarily with a probability $p$.
- The hyperparameter $p$ is called the dropout rate.
At each training step, each neuron drops out temporarily with a probability $p$.

- The hyperparameter $p$ is called the dropout rate.
- A neuron will be entirely ignored during this training step.
At each training step, each neuron drops out temporarily with a probability $p$.

- The hyperparameter $p$ is called the dropout rate.
- A neuron will be entirely ignored during this training step.
- It may be active during the next step.
At each training step, each neuron drops out temporarily with a probability $p$.

- The hyperparameter $p$ is called the dropout rate.
- A neuron will be entirely ignored during this training step.
- It may be active during the next step.
- Exclude the output neurons.
At each training step, each neuron drops out temporarily with a probability $p$.

- The hyperparameter $p$ is called the dropout rate.
- A neuron will be entirely ignored during this training step.
- It may be active during the next step.
- Exclude the output neurons.

After training, neurons don’t get dropped anymore.
Each neuron can be either present or absent.

$2^N$ possible networks, where $N$ is the total number of droppable neurons.

- $N = 4$ in this figure.
Data Augmentation

- One way to make a model generalize better is to train it on more data.
- This will reduce overfitting.
Data Augmentation

- One way to make a model generalize better is to train it on more data.
- This will reduce overfitting.
- Create fake data and add it to the training set.
Batch Size
Training Deep Neural Networks

- Computationally intensive
- Time consuming
Why?

- Massive amount of training dataset
- Large number of parameters
Accuracy vs. Data/Model Size

1980s and 1990s

Accuracy

Scale (data size, model size)

neural networks

other approaches

[Jeff Dean at AI Frontiers: Trends and Developments in Deep Learning Research]
Accuracy vs. Data/Model Size

1980s and 1990s

Accuracy

more compute

neural networks

other approaches

Scale (data size, model size)
Accuracy vs. Data/Model Size

Now

Accuracy

Scale (data size, model size)

more compute

neural networks

other approaches

[Jeff Dean at AI Frontiers: Trends and Developments in Deep Learning Research]
Scale Matters

Scalability
Distributed Gradient Descent (1/2)

- Replicate a whole model on every device.
- Each device has model replica with a copy of model parameters.

Distributed Gradient Descent (2/2)

- **Parameter Server (PS):** maintains global model.

Distributed Gradient Descent (2/2)

- **Parameter Server (PS):** maintains global model.
- Once each device completes processing, the weights are transferred to **PS**, which aggregates all the gradients.

Distributed Gradient Descent (2/2)

- **Parameter Server (PS):** maintains global model.
- Once each device completes processing, the weights are transferred to **PS**, which aggregates all the gradients.
- The PS, then, sends back the results to each device.

Batch Size vs. Number of GPUs

- \( w \leftarrow w - \eta \frac{1}{|\beta|} \sum_{x \in \beta} \nabla l(x, w) \)

- The more samples processed during each batch, the faster a training job will complete.

- E.g., ImageNet + ResNet-50

[https://medium.com/@emwatz/lessons-for-improving-training-performance-part-1-b5efd0f0dcea]
**Batch Size vs. Number of GPUs**

- \( w \leftarrow w - \eta \frac{1}{|\beta|} \sum_{x \in \beta} \nabla l(x, w) \)
- The more samples processed during each batch, the faster a training job will complete.

![Batch size 64 v. 256 (FP16)](https://medium.com/@emwatz/lessons-for-improving-training-performance-part-1-b5efd0f0dcea)
Batch Size vs. Number of GPUs

- $\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{1}{|\beta|} \sum_{x \in \beta} \nabla l(x, \mathbf{w})$

- The more samples processed during each batch, the faster a training job will complete.

- E.g., ImageNet + ResNet-50

[https://medium.com/@emwatz/lessons-for-improving-training-performance-part-1-b5efd0f0dce]
ResNet-32 on Titan X GPU

![Graph showing batch size vs. time to accuracy with batch sizes 32, 64, 128, 256, 512, 1024 and corresponding time in seconds for TensorFlow.]

[Peter Pietzuch - Imperial College London]
Batch Size vs. Validation Error

[Image: Graph showing the relationship between mini-batch size and ImageNet top-1 validation error.]

[Goyal et al., Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, 2018]
Improve the Validation Error
Improve the Validation Error

- Scaling learning rate
- Batch normalization
- Label smoothing
- Momentum
Scaling Learning Rate

- $\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{1}{|\beta|} \sum_{x \in \beta} \nabla l(x, \mathbf{w})$.

- Linear scaling: multiply the learning rate by $k$, when the mini batch size is multiplied by $k$.

- Constant warmup: start with a small learning rate for few epochs, and then increase the learning rate to $k$ times learning rate.

- Gradual warmup: start with a small learning rate, and then gradually increase it by a constant for each epoch till it reaches $k$ times learning rate.

[Goyal et al., Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, 2018]
Scaling Learning Rate

- \( w \leftarrow w - \eta \frac{1}{|\beta|} \sum_{x \in \beta} \nabla l(x, w) \).

- **Linear scaling**: multiply the **learning rate** by \( k \), when the **mini batch size** is multiplied by \( k \).
Scaling Learning Rate

- \( w \leftarrow w - \eta \frac{1}{|\beta|} \sum_{x \in \beta} \nabla l(x, w) \).

- **Linear scaling**: multiply the learning rate by \( k \), when the mini batch size is multiplied by \( k \).

- **Constant warmup**: start with a small learning rate for few epochs, and then increase the learning rate to \( k \) times learning rate.

[Goyal et al., Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, 2018]
Scaling Learning Rate

- \( w \leftarrow w - \eta \frac{1}{|\beta|} \sum_{x \in \beta} \nabla l(x, w) \).

- **Linear scaling:** multiply the learning rate by \( k \), when the mini batch size is multiplied by \( k \).

- **Constant warmup:** start with a small learning rate for few epochs, and then increase the learning rate to \( k \) times learning rate.

- **Gradual warmup:** start with a small learning rate, and then gradually increase it by a constant for each epoch till it reaches \( k \) times learning rate.

---

[47x476] [Goyal et al., Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, 2018]
Scaling Learning Rate

- \( \mathbf{w} \leftarrow \mathbf{w} - \eta \frac{1}{|\beta|} \sum_{x \in \beta} \nabla l(x, \mathbf{w}) \).

- **Linear scaling**: multiply the learning rate by \( k \), when the mini batch size is multiplied by \( k \).

- **Constant warmup**: start with a small learning rate for few epochs, and then increase the learning rate to \( k \) times learning rate.

- **Gradual warmup**: start with a small learning rate, and then gradually increase it by a constant for each epoch till it reaches \( k \) times learning rate.

[Goyal et al., Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, 2018]
Batch Normalization (1/2)

- Changes in minibatch size change the underlying loss function being optimized.
- Changes in minibatch size change the underlying loss function being optimized.
- Batch Normalization computes statistics along the minibatch dimension.
Batch Normalization (1/2)

- Changes in minibatch size change the underlying loss function being optimized.
- Batch Normalization computes statistics along the minibatch dimension.

\[
\mu_\beta = \frac{1}{|\beta|} \sum_{x \in \beta} x \\
\sigma^2_\beta = \frac{1}{|\beta|} \sum_{x \in \beta} (x - \mu_\beta)^2
\]
Batch Normalization (2/2)

▶ Zero-centering and normalizing the inputs, then scaling and shifting the result.

\[
\hat{x} = \frac{x - \mu_{\beta}}{\sqrt{\sigma_{\beta}^2 + \epsilon}}
\]

\[
z = \alpha \hat{x} + \gamma
\]

▶ \(\hat{x}\): the zero-centered and normalized input.
▶ \(z\): the output of the BN operation, which is a scaled and shifted version of the inputs.
▶ \(\alpha\): the scaling parameter vector for the layer.
▶ \(\gamma\): the shifting parameter (offset) vector for the layer.
▶ \(\epsilon\): a tiny number to avoid division by zero.
Label Smoothing

- A generalization technique.
- Replaces one-hot encoded label vector $\mathbf{y}_{\text{hot}}$ with a mixture of $\mathbf{y}_{\text{hot}}$ and the uniform distribution.

$$\mathbf{y}_{\text{ls}} = (1 - \alpha) \mathbf{y}_{\text{hot}} + \alpha/K$$

- $K$ is the number of label classes, and $\alpha$ is a hyperparameter.

[Shallue et al., Measuring the Effects of Data Parallelism on Neural Network Training, 2019]
Momentum (1/3)

- Regular gradient descent optimization: $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla J(\mathbf{w})$
Momentum (1/3)

- Regular gradient descent optimization: \( \mathbf{w} \leftarrow \mathbf{w} - \eta \nabla J(\mathbf{w}) \)
- At each iteration, *momentum optimization* adds the local gradient to the *momentum vector* \( \mathbf{m} \).

\[
\mathbf{m} \leftarrow \beta \mathbf{m} + \eta \nabla J(\mathbf{w}) \\
\mathbf{w} \leftarrow \mathbf{w} - \mathbf{m}
\]

Nesterov momentum measure the gradient of the cost function slightly ahead in the direction of the momentum.

\[ m = \beta m + \eta \nabla J(w + \beta m) \]

\[ w \leftarrow w - m \]
Momentum (3/3)

(a) Simple CNN on MNIST  
(b) Transformer Shallow on LM1B  
(c) ResNet-8 on CIFAR-10

[Shallue et al., Measuring the Effects of Data Parallelism on Neural Network Training, 2019]
CROSSBOW: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers
How to design a deep learning system that scales training with multiple GPUs, even when the preferred batch size is small?
(1) How to increase efficiency with small batches?

(2) How to synchronise model replicas?
Problem: Small Batches

- Small batch sizes underutilise GPUs.
Problem: Small Batches

- **Small** batch sizes **underutilise** GPUs.
- **One batch** per GPU: **not enough data** and instruction parallelism for every operator.

[Peter Pietzuch - Imperial College London]
Idea: Multiple Replicas Per GPU

- Train **multiple model replicas** per GPU.
- A **learner** is an entity that trains a **single model replica** independently with a given batch size.

[Peter Pietzuch - Imperial College London]
Idea: Multiple Replicas Per GPU

- Train multiple model replicas per GPU.
- A learner is an entity that trains a single model replica independently with a given batch size.

- But, now we must synchronise a large number of model replicas.
Problem: Similar Starting Point

- All learners always start from the same point.
- Limited exploration of parameter space.
Idea: Independent Replicas

- Maintain **independent** model replicas.
- **Increased exploration** of space through parallelism.
- **Each model replica** uses **small batch size**.

[Peter Pietzuch - Imperial College London]
Crossbow: Synchronous Model Averaging

- Allow learners to diverge, but correct trajectories based on average model.
- Accelerate average model trajectory with momentum to find minima faster.
GPUs with Synchronous Model Averaging

- Synchronously apply corrections to model replicas.
GPUs with Synchronous Model Averaging

- Ensures **consistent view** of **average model**.
- Takes **GPU bandwidth** into account during synchronisation.
(1) How to increase efficiency with small batches?

Train multiple model replicas per GPU

(2) How to synchronise model replicas?

Use synchronous model averaging

[Peter Pietzuch - Imperial College London]
Summary
Summary

- Stochastic Gradient Descent (SGD)
- Generalization
  - Regularization
  - Max-norm
  - Dropout
- Distributed SGD
- Batch size
  - Scaling learning rate
  - Batch normalization
  - Label smoothing
  - Momentum
- Crossbow
Reference

- P. Goyal et al., Accurate, large minibatch sgd: Training imagenet in 1 hour, 2017
- C. Shallue et al., Measuring the effects of data parallelism on neural network training, 2018
- A. Koliouisis et al. CROSSBOW: scaling deep learning with small batch sizes on multi-gpu servers, 2019
Questions?