Batch Normalization

Shusen Wang

Feature Scaling for Linear Models

Why Feature Scaling?

People's feature vectors: $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^2$.

- The 1st dimension is one person's income (in dollars).
  - Assume it is randomly from the Gaussian distribution $N(3000,\ 400^2)$.
- The 2nd dimension is one person's height (in inch).
  - Assume it is randomly from the Gaussian distribution $N(69,\ 3^2)$.

- Hessian matrix of least squares regression model:

$$\mathbf{H} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^T = \begin{bmatrix} 9137.3 & 206.6 \\ 206.6 & 4.8 \end{bmatrix} \times 10^3.$$

- Condition number: $\frac{\lambda_{\max}(\mathbf{H})}{\lambda_{\min}(\mathbf{H})} = 9.2 \times 10^4$.

Bad condition number means slow convergence of gradient descent!

Why Feature Scaling?

People's feature vectors: $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^2$.

- The 1st dimension is one person's income (in thousand dollars).
  - Assume it is randomly from the Gaussian distribution $N(3,\ 0.4^2)$.
- The 2nd dimension is one person's height (in foot).
  - Assume it is randomly from the Gaussian distribution $N(5.75,\ \ 0.25^2)$.
- Change metric.
- Hessian matrix of linear regression:

$$\mathbf{H} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^T = \begin{bmatrix} 9.1 & 17.2 \\ 17.2 & 33.1 \end{bmatrix}.$$

- Condition number: $\frac{\lambda_{\max}(\mathbf{H})}{\lambda_{\min}(\mathbf{H})} = 281.7.$

Feature Scaling for 1D Data

Assume the samples $x_1, \cdots, x_n$ are one-dimensional.

- **Min-max normalization**: $x_i' = \frac{x_i - \min(x_i)}{\max(x_i) - \min(x_i)}$.
- The samples $x_1', \cdots, x_n'$ are in $[0, 1]$; zero and one are attained.

- **Standardization**: $x_i' = \frac{x_i - \widehat{\mu}}{\widehat{\sigma}}$.
  - $\hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} x_i$ is the sample mean.
  - $\hat{\sigma}^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \hat{\mu})^2$ is the sample variance.

Feature Scaling for High-Dim Data

- Perform feature scaling for every feature independently.
  - E.g., when scaling the “height” feature, ignore the “income” feature.

```
# Min-Max Normalization
import numpy

d = x.shape[1]
xmin = numpy.min(x, axis=0).reshape(1, d)
xmax = numpy.max(x, axis=0).reshape(1, d)
xnew = (x - xmin) / (xmax - xmin)
```

Feature Scaling for High-Dim Data

- Independently perform feature scaling for every feature.
  - E.g., when scaling the “height” feature, ignore the “income” feature.

```
# Standardization
import numpy

d = x.shape[1]
mu = numpy.mean(x, axis=0).reshape(1, d)
sig = numpy.std(x, axis=0).reshape(1, d)
xnew = (x - mu) / sig
```

Batch Normalization

Batch Normalization: Standardization of Hidden Layers

- Let $\mathbf{x}^{(k)} \in \mathbb{R}^d$ be the output of the k-th hidden layer.
- $\hat{\boldsymbol{\mu}} \in \mathbb{R}^d$: sample mean of $\mathbf{x}^{(k)}$ evaluated on a batch of samples.
- $\hat{\boldsymbol{\sigma}} \in \mathbb{R}^d$: sample std of $\mathbf{x}^{(k)}$ evaluated on a batch of samples.

- Standardization: $z_j^{(k)} = \frac{x_j^{(k)} - \hat{\mu}_j}{\hat{\sigma}_j + 0.001}$, for $j = 1, \cdots, d$.

Batch Normalization: Standardization of Hidden Layers

- Let $\mathbf{x}^{(k)} \in \mathbb{R}^d$ be the output of the k-th hidden layer.
- $\hat{\boldsymbol{\mu}} \in \mathbb{R}^d$: sample mean of $\mathbf{x}^{(k)}$ evaluated on a batch of samples.
- $\hat{\boldsymbol{\sigma}} \in \mathbb{R}^d$: sample std of $\mathbf{x}^{(k)}$ evaluated on a batch of samples.
- $\boldsymbol{\gamma} \in \mathbb{R}^d$: scaling parameter (trainable).
- $\boldsymbol{\beta} \in \mathbb{R}^d$: shifting parameter (trainable).
- Standardization: $z_j^{(k)} = \frac{x_j^{(k)} - \hat{\mu}_j}{\hat{\sigma}_j + 0.001}$, for $j = 1, \cdots, d$.
- Scale and shift: $x_j^{(k+1)} = z_j^{(k)} \cdot \gamma_j + \beta_j$, for $j = 1, \cdots, d$.

Batch Normalization: Standardization of Hidden Layers

- Let $\mathbf{x}^{(k)} \in \mathbb{R}^d$ be the output of the k-th hidden layer.
- $\hat{\boldsymbol{\mu}} \in \mathbb{R}^d$: **Non-trainable**. Just record them in the forward pass;
- $\hat{\boldsymbol{\sigma}} \in \mathbb{R}^d$: use them in the backpropagation.
- $\boldsymbol{\gamma} \in \mathbb{R}^d$: scaling parameter (trainable).
- $\boldsymbol{\beta} \in \mathbb{R}^d$: shifting parameter (trainable).
- Standardization: $z_j^{(k)} = \frac{x_j^{(k)} - \hat{\mu}_j}{\hat{\sigma}_j + 0.001}$, for $j = 1, \cdots, d$.
- Scale and shift: $x_j^{(k+1)} = z_j^{(k)} \cdot \gamma_j + \beta_j$, for $j = 1, \cdots, d$.

Backpropagation for Batch Normalization Layer

- Standardization: $z^{(k)} = \frac{x^{(k)} - \widehat{\mu}_j}{\widehat{\sigma}_j + 0.001}$, for $j = 1, \cdots, d$.
- Scale and shift: $x^{(k+1)} = z^{(k)} \cdot \gamma_j + \beta_j$, for $j = 1, \cdots, d$.

We know $\frac{\partial L}{\partial x^{(k+1)}}$ from the backpropagation (from the top to $x^{(k+1)}$.)

- Use $\frac{\partial L}{\partial \gamma_j} = \frac{\partial L}{\partial x^{(k+1)}} \frac{\partial x^{(k+1)}}{\partial \gamma_j} = \frac{\partial L}{\partial x^{(k+1)}} z^{(k)}$ to update γ_j ;
- Use $\frac{\partial L}{\partial \beta_j} = \frac{\partial L}{\partial x^{(k+1)}} \frac{\partial x^{(k+1)}}{\partial \beta_j} = \frac{\partial L}{\partial x^{(k+1)}}$ to update β_j .

Backpropagation for Batch Normalization Layer

- Standardization: $z^{(k)} = \frac{\boxed{x^{(k)}} - \hat{\mu}_j}{\hat{\sigma}_j + 0.001}$, for $j = 1, \cdots, d$.
- Scale and shift: $x^{(k+1)} = z^{(k)} \cdot \gamma_j + \beta_j$, for $j = 1, \cdots, d$.

We know $\frac{\partial L}{\partial x^{(k+1)}}$ from the backpropagation (from the top to $x^{(k+1)}$.)

Compute $\boxed{\frac{\partial L}{\partial z^{(k)}}} = \frac{\partial L}{\partial x^{(k+1)}} \frac{\partial x^{(k+1)}}{\partial z^{(k)}} = \frac{\partial L}{\partial x^{(k+1)}} \gamma_j$.

Compute $\boxed{\frac{\partial L}{\partial x^{(k)}}} = \boxed{\frac{\partial L}{\partial z^{(k)}}} \frac{\partial z^{(k)}}{\partial x^{(k)}} = \boxed{\frac{\partial L}{\partial z^{(k)}}} \frac{1}{\hat{\sigma}_j + 0.001}$ and pass it to the lower layers.

Batch Normalization Layer in Keras

Batch Normalization Layer

- Let $\mathbf{x}^{(k)} \in \mathbb{R}^d$ be the output of the k-th hidden layer.
- $\hat{\boldsymbol{\mu}}, \ \hat{\boldsymbol{\sigma}} \in \mathbb{R}^d$: **non-trainable** parameters.
- $\boldsymbol{\gamma}, \ \boldsymbol{\beta} \in \mathbb{R}^d$: **trainable** parameters.
- Standardization: $z_j^{(k)} = \frac{x_j^{(k)} - \hat{\mu}_j}{\hat{\sigma}_j + 0.001}$, for $j = 1, \cdots, d$.
- Scale and shift: $x_j^{(k+1)} = z_j^{(k)} \cdot \gamma_j + \beta_j$, for $j = 1, \cdots, d$.

Batch Normalization Layer

- Let $\mathbf{x}^{(k)} \in \mathbb{R}^d$ be the output of the k-th hidden layer.
- $\hat{\boldsymbol{\mu}}$, $\hat{\boldsymbol{\sigma}} \in \mathbb{R}^d$: **non-trainable** parameters.
- $\boldsymbol{\gamma}$, $\boldsymbol{\beta} \in \mathbb{R}^d$: **trainable** parameters.

Difficulty: There are $4d$ parameters which must be stored in memory. d can be very large!

Example:

- The 1st Conv Layer in VGG16 Net outputs a $150 \times 150 \times 64$ tensor.
- The number of parameters in a single Batch Normalization Layer would be $4d = 1.44\text{M}$.

Batch Normalization Layer

Solution:

- Make the 4 parameters $1 \times 1 \times 64$, instead of $150 \times 150 \times 64$.
- How?
- A scalar parameter for a slice (e.g., a 150×150 matrix) of the tensor.
- Of course, you can make the parameters $150 \times 1 \times 1$ or $1 \times 150 \times 1$.

d can be very large!

Example:

- The 1st Conv Layer in VGG16 Net outputs a $150 \times 150 \times 64$ tensor.
- The number of parameters in a single Batch Normalization Layer would be $4d = 1.44\text{M}$.

CNN for Digit Recognition

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Conv2D(10, (5, 5), input_shape=(28, 28, 1)))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(20, (5, 5)))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(100))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
model.add(layers.Dense(10, activation='softmax'))
```

CNN for Digit Recognition

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 24, 24, 10)        260
_________________________________________________________________
batch_normalization_1 (Batch (None, 24, 24, 10)        40
_________________________________________________________________
activation_1 (Activation)    (None, 24, 24, 10)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 10)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 20)          5020
_________________________________________________________________
batch_normalization_2 (Batch (None, 8, 8, 20)          80
_________________________________________________________________
activation_2 (Activation)    (None, 8, 8, 20)          0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 20)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 320)               0
_________________________________________________________________
dense_1 (Dense)              (None, 100)               32100
_________________________________________________________________
batch_normalization_3 (Batch (None, 100)               400
_________________________________________________________________
activation_3 (Activation)    (None, 100)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 38,910
Trainable params: 38,650
Non-trainable params: 260
```

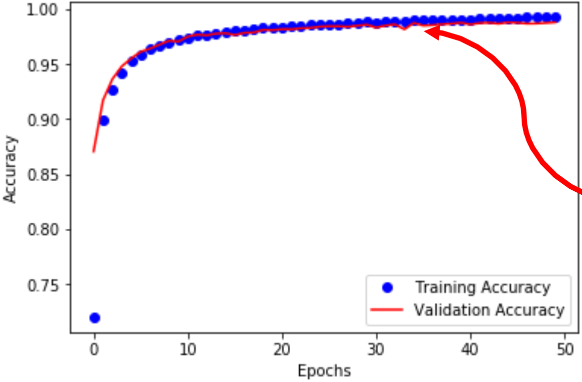
CNN for Digit Recognition

Train the model (**with** Batch Normalization) on MNIST ($n = 50{,}000$).

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/3
50000/50000 [==============================] - 29s 580us/step -
loss: 0.1599 - acc: 0.9595 - val_loss: 0.1165 - val_acc: 0.9644
Epoch 2/3
50000/50000 [==============================] - 26s 516us/step -
loss: 0.0468 - acc: 0.9858 - val_loss: 0.0562 - val_acc: 0.9822
Epoch 3/3
50000/50000 [==============================] - 25s 508us/step -
loss: 0.0325 - acc: 0.9902 - val_loss: 0.0494 - val_acc: 0.9832
```

CNN for Digit Recognition

Train the model (**without** Batch Normalization) on MNIST ($n = 50{,}000$).



Without Batch Normalization, it takes 10x more epochs to converge.