ISMT S-136 Time Series Analysis with Python

Harvard Summer School

Dmitry Kurochkin

Summer 2021
Lecture 11

Contents

1. Recurrent Neural Networks (RNN)
   - Recurrent Neuron and Layer of Recurrent Neurons
   - Memory Cells
   - Long Short-Term Memory (LSTM) Cell
   - Gated Recurrent Unit (GRU) Cell

2. Unstable Gradients
   - Vanishing/Exploding Gradients Problems
   - Techniques to Alleviate the Unstable Gradient Problems

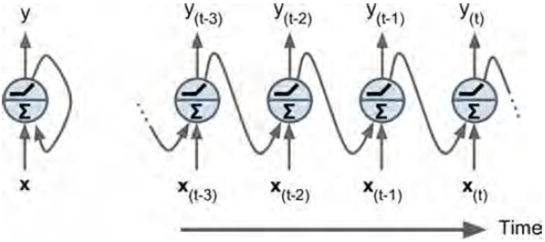
3. Neural Network Optimization Algorithms
   - Objective (Cost) Function
   - SGD, mini-batch GD, and GD Optimization
   - Forward Propagation and Backpropagation
   - Momentum Optimization, NAG, AdaGrad, RMSProp, and Adam

Contents

1. Recurrent Neural Networks (RNN)
   - Recurrent Neuron and Layer of Recurrent Neurons
   - Memory Cells
   - Long Short-Term Memory (LSTM) Cell
   - Gated Recurrent Unit (GRU) Cell
2. Unstable Gradients
   - Vanishing/Exploding Gradients Problems
   - Techniques to Alleviate the Unstable Gradient Problems
3. Neural Network Optimization Algorithms
   - Objective (Cost) Function
   - SGD, mini-batch GD, and GD Optimization
   - Forward Propagation and Backpropagation
   - Momentum Optimization, NAG, AdaGrad, RMSProp, and Adam

RNN: Recurrent Neuron

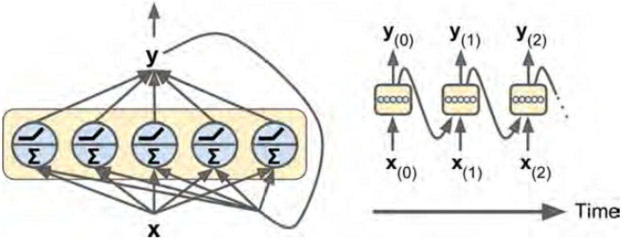
Recurrent Neuron:



Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

RNN: Layer of Recurrent Neurons

Layer of Recurrent Neurons:



Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Layer of Recurrent Neurons: Keras

Simple RNN in Keras:

```
n_features = 2
n_timesteps = 200

model = models.Sequential()
model.add(layers.SimpleRNN(3, activation='relu', input_shape=(n_timesteps,n_features)))
model.add(layers.Dense(1, activation='linear'))

model.summary()
```

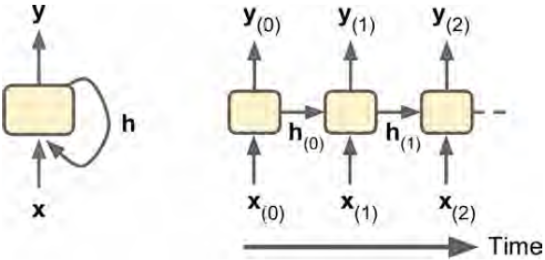
```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_6 (SimpleRNN)     (None, 3)                 18
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 4
=================================================================
Total params: 22
Trainable params: 22
Non-trainable params: 0
```

Contents

1. Recurrent Neural Networks (RNN)
   - Recurrent Neuron and Layer of Recurrent Neurons
   - Memory Cells
   - Long Short-Term Memory (LSTM) Cell
   - Gated Recurrent Unit (GRU) Cell
2. Unstable Gradients
   - Vanishing/Exploding Gradients Problems
   - Techniques to Alleviate the Unstable Gradient Problems
3. Neural Network Optimization Algorithms
   - Objective (Cost) Function
   - SGD, mini-batch GD, and GD Optimization
   - Forward Propagation and Backpropagation
   - Momentum Optimization, NAG, AdaGrad, RMSProp, and Adam

RNN: Memory Cells

Layer of Recurrent Neurons:



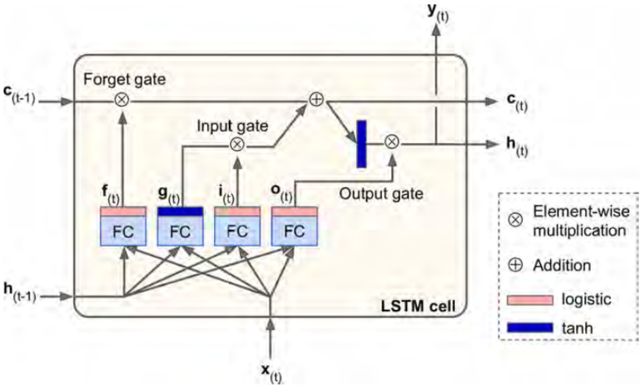
Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Contents

1. Recurrent Neural Networks (RNN)
   - Recurrent Neuron and Layer of Recurrent Neurons
   - Memory Cells
   - Long Short-Term Memory (LSTM) Cell
   - Gated Recurrent Unit (GRU) Cell
2. Unstable Gradients
   - Vanishing/Exploding Gradients Problems
   - Techniques to Alleviate the Unstable Gradient Problems
3. Neural Network Optimization Algorithms
   - Objective (Cost) Function
   - SGD, mini-batch GD, and GD Optimization
   - Forward Propagation and Backpropagation
   - Momentum Optimization, NAG, AdaGrad, RMSProp, and Adam

Long Short-Term Memory (LSTM) Cell

LSTM Cell:



Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Long Short-Term Memory (LSTM) Cell

LSTM Cell Model:

$$
\begin{aligned}
\mathbf{i}_{(t)} &= \sigma\left(\mathbf{W}_{xi}{}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}{}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_i\right)\\
\mathbf{f}_{(t)} &= \sigma\left(\mathbf{W}_{xf}{}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}{}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_f\right)\\
\mathbf{o}_{(t)} &= \sigma\left(\mathbf{W}_{xo}{}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{ho}{}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_o\right)\\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_g\right)\\
\mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)}\\
\mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right)
\end{aligned}
$$

- $\mathbf{W}_{xi}$, $\mathbf{W}_{xf}$, $\mathbf{W}_{xo}$, $\mathbf{W}_{xg}$ are the weight matrices of each of the four layers for their connection to the input vector $\mathbf{x}_{(t)}$.
- $\mathbf{W}_{hi}$, $\mathbf{W}_{hf}$, $\mathbf{W}_{ho}$, and $\mathbf{W}_{hg}$ are the weight matrices of each of the four layers for their connection to the previous short-term state $\mathbf{h}_{(t-1)}$.
- $\mathbf{b}_i$, $\mathbf{b}_f$, $\mathbf{b}_o$, and $\mathbf{b}_g$ are the bias terms for each of the four layers. Note that TensorFlow initializes $\mathbf{b}_f$ to a vector full of 1s instead of 0s. This prevents forgetting everything at the beginning of training.

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Applications of LSTM Cells

- greatly improved speech recognition on over 4 billion Android phones (since mid 2015)
- greatly improved machine translation through Google Translate (since Nov 2016)
- greatly improved machine translation through Facebook (over 4 billion LSTMbased translations per day as of 2017)
- Siri and Quicktype on almost 2 billion iPhones (since 2016)
- generating answers by Amazon's Alexa and numerous other similar applications.

Long Short-Term Memory (LSTM) Cell: Keras

LSTM in Keras:

```
n_features = 2
n_timesteps = 200

model = models.Sequential()
model.add(LSTM(16, activation='relu', input_shape=(n_timesteps,n_features)))
model.add(layers.Dense(1, activation='linear'))

model.summary()
```

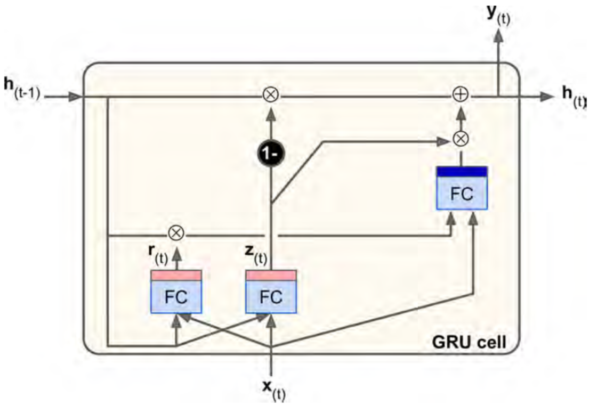
```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                 (None, 16)               1216
_________________________________________________________________
dense_7 (Dense)               (None, 1)                17
=================================================================
Total params: 1,233
Trainable params: 1,233
Non-trainable params: 0
```

Contents

1. Recurrent Neural Networks (RNN)
   - Recurrent Neuron and Layer of Recurrent Neurons
   - Memory Cells
   - Long Short-Term Memory (LSTM) Cell
   - Gated Recurrent Unit (GRU) Cell
2. Unstable Gradients
   - Vanishing/Exploding Gradients Problems
   - Techniques to Alleviate the Unstable Gradient Problems
3. Neural Network Optimization Algorithms
   - Objective (Cost) Function
   - SGD, mini-batch GD, and GD Optimization
   - Forward Propagation and Backpropagation
   - Momentum Optimization, NAG, AdaGrad, RMSProp, and Adam

Gated Recurrent Unit (GRU) Cell

GRU Cell:



Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Gated Recurrent Unit (GRU) Cell

GRU Cell Model:

$$
\begin{aligned}
\mathbf{z}_{(t)} &= \sigma\left(\mathbf{W}_{xz}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{T} \cdot \mathbf{h}_{(t-1)}\right) \\
\mathbf{r}_{(t)} &= \sigma\left(\mathbf{W}_{xr}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{T} \cdot \mathbf{h}_{(t-1)}\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot \left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right)\right) \\
\mathbf{h}_{(t)} &= \left(1 - \mathbf{z}_{(t)}\right) \otimes \tanh\left(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{z}_{(t)} \otimes \mathbf{g}_{t}\right)
\end{aligned}
$$

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Contents

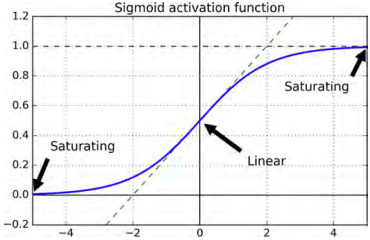
1. Recurrent Neural Networks (RNN)
   - Recurrent Neuron and Layer of Recurrent Neurons
   - Memory Cells
   - Long Short-Term Memory (LSTM) Cell
   - Gated Recurrent Unit (GRU) Cell
2. Unstable Gradients
   - Vanishing/Exploding Gradients Problems
   - Techniques to Alleviate the Unstable Gradient Problems
3. Neural Network Optimization Algorithms
   - Objective (Cost) Function
   - SGD, mini-batch GD, and GD Optimization
   - Forward Propagation and Backpropagation
   - Momentum Optimization, NAG, AdaGrad, RMSProp, and Adam

Vanishing/Exploding Gradients Problems

Unstable gradients:

1. *Vanishing gradients* problem: Given current weights $\boldsymbol{w}$ of the NN and inputs (data), the gradient of the activation function may be very small resulting in the corresponding weights virtually unchanged during the iterations / updates.
2. *Exploding gradients* problem: The weights may one the contrary blow up - this problem is mostly encountered in recurrent neural networks.

Sigmoid function:



Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Contents

1. Recurrent Neural Networks (RNN)
   - Recurrent Neuron and Layer of Recurrent Neurons
   - Memory Cells
   - Long Short-Term Memory (LSTM) Cell
   - Gated Recurrent Unit (GRU) Cell
2. Unstable Gradients
   - Vanishing/Exploding Gradients Problems
   - Techniques to Alleviate the Unstable Gradient Problems
3. Neural Network Optimization Algorithms
   - Objective (Cost) Function
   - SGD, mini-batch GD, and GD Optimization
   - Forward Propagation and Backpropagation
   - Momentum Optimization, NAG, AdaGrad, RMSProp, and Adam

Techniques to Alleviate the Unstable Gradient Problems

Ways to resolve the problems:

1. "Proper" initialization of weights: special initial distribution, reusing pretrained layers, etc.
2. Nonsaturating activations functons: Leaky ReLU, exponential LU (ELU), etc.
3. Batch normalization (BN): scale inputs before each layer during training (two more parameters)
4. Gradient clipping: set a threshold for the gradient

Contents

1. Recurrent Neural Networks (RNN)
   - Recurrent Neuron and Layer of Recurrent Neurons
   - Memory Cells
   - Long Short-Term Memory (LSTM) Cell
   - Gated Recurrent Unit (GRU) Cell
2. Unstable Gradients
   - Vanishing/Exploding Gradients Problems
   - Techniques to Alleviate the Unstable Gradient Problems
3. Neural Network Optimization Algorithms
   - Objective (Cost) Function
   - SGD, mini-batch GD, and GD Optimization
   - Forward Propagation and Backpropagation
   - Momentum Optimization, NAG, AdaGrad, RMSProp, and Adam

Objective (Cost) Function

Suppose we want to train a supervised model (e.g., Neural Network) using a set of observations:

$$(\boldsymbol{x}_1, \boldsymbol{y}_1), (\boldsymbol{x}_2, \boldsymbol{y}_2), (\boldsymbol{x}_3, \boldsymbol{y}_3), \ldots, (\boldsymbol{x}_m, \boldsymbol{y}_m)$$

then we define the objective (or cost) function as mean loss:

$$J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} L^{(i)}(\boldsymbol{w}),$$

where

$$L^{(i)}(\boldsymbol{w}) = L(\underbrace{\hat{\boldsymbol{y}}^{(i)}(\boldsymbol{w})}_{\text{prediction}}, \underbrace{\boldsymbol{y}^{(i)}}_{\text{observed}})$$

is the loss associated with a single observation i.

Objective (Cost) Function

The list of the most common cost functions:

- Mean Squared Error:

$$J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{M}(\hat{y}_j^{(i)} - y_j^{(i)})^2$$

- Mean Absolute Error:

$$J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}\left(\sum_{j=1}^{M}(\hat{y}_j^{(i)} - y_j^{(i)})^2\right)^{\frac{1}{2}}$$

- Cross-Entropy:

$$J(\boldsymbol{w}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{M} y_j^{(i)} \ln \hat{y}_j^{(i)}$$

Contents

1. Recurrent Neural Networks (RNN)
   - Recurrent Neuron and Layer of Recurrent Neurons
   - Memory Cells
   - Long Short-Term Memory (LSTM) Cell
   - Gated Recurrent Unit (GRU) Cell
2. Unstable Gradients
   - Vanishing/Exploding Gradients Problems
   - Techniques to Alleviate the Unstable Gradient Problems
3. Neural Network Optimization Algorithms
   - Objective (Cost) Function
   - SGD, mini-batch GD, and GD Optimization
   - Forward Propagation and Backpropagation
   - Momentum Optimization, NAG, AdaGrad, RMSProp, and Adam

SGD, mini-batch GD, and GD Optimization: 'sgd'

The SGD, mini-batch GD, and GD Optimization (with learning rate α) are all defined as follows:

$$\boldsymbol{w} := \boldsymbol{w} - \alpha \underbrace{\frac{1}{s} \sum_{i=1}^{s} \nabla L^{(i)}(\boldsymbol{w})}_{\approx \nabla J(\boldsymbol{w})},$$

where $L^{(i)}(\boldsymbol{w})$ is based on one observation i and

- $s = 1$ in case of Stochastic Gradient Descent (SGD)
- $1 < s < m$ in case of mini-batch Gradient Descent (mini-batch GD)
- $s = m$ in case of Gradient Descent (GD)

Here, m denotes the total number of observations in the data set.

SGD, mini-batch GD, and GD Optimization

Example: Mini-batch GD with $s = 128$ and $\alpha = 0.01$.

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 35
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='sgd')

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

SGD, mini-batch GD, and GD Optimization

Example: Mini-batch GD with $s = 128$ and $\alpha = 0.05$.

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 35
model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
              optimizer=keras.optimizers.SGD(lr=0.05))

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

Contents

1 Recurrent Neural Networks (RNN)
- Recurrent Neuron and Layer of Recurrent Neurons
- Memory Cells
- Long Short-Term Memory (LSTM) Cell
- Gated Recurrent Unit (GRU) Cell

2 Unstable Gradients
- Vanishing/Exploding Gradients Problems
- Techniques to Alleviate the Unstable Gradient Problems

3 Neural Network Optimization Algorithms
- Objective (Cost) Function
- SGD, mini-batch GD, and GD Optimization
- Forward Propagation and Backpropagation
- Momentum Optimization, NAG, AdaGrad, RMSProp, and Adam

Forward Propagation

Let's again consider a Neural Network (NN) with

n inputs,
M outputs and
1 hidden layer with H neurons.

The Forward Propagation is then

$$\hat{y}_m = \sigma_m^{(2)} \left(\sum_{h=0}^{H} w_{hm}^{(2)} \underbrace{\sigma_h^{(1)} \left(\sum_{j=0}^{n} w_{jh}^{(1)} x_j \right)}_{\doteq u_h} \right).$$

Backpropagation

Given an observation $(\boldsymbol{x}, \boldsymbol{y})$, assume we want to minimize the Mean Squared Error Loss

$$L(\boldsymbol{w}) = \sum_{m=1}^{M} \left(\hat{y}_m - y_m\right)^2 ,$$

where

$$\hat{y}_m = \sigma_m^{(2)} \left(\sum_{h=0}^{H} w_{hm}^{(2)} \underbrace{\sigma_h^{(1)} \left(\sum_{j=0}^{n} w_{jh}^{(1)} x_j \right)}_{\doteq u_h} \right) .$$

Then need to compute $\frac{L(\boldsymbol{w})}{\partial w_{hm}^{(2)}}$ and $\frac{L(\boldsymbol{w})}{\partial w_{jh}^{(1)}}$.

But we know derivatives of $\sigma_m^{(2)}$ and $\sigma_h^{(1)}$ exactly!

Also, we have $\sum_{j=0}^{n} w_{jh}^{(1)} x_j$, u_h, $\sum_{h=0}^{H} w_{hm}^{(2)} u_h$, and $\hat{y}_m$ computed during forward propagation!

Example of Forward Propagation/Backpropagation

Let's consider the following Neural Network:

input layer	hidden layer	output layer
x_1	$u_1 = f(\underbrace{w_{01}^{(1)} + w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2}_{z_1^{(1)}})$	$\hat{y} = f(\underbrace{w_0^{(2)} + w_1^{(2)} u_1 + w_2^{(2)} u_2}_{z^{(2)}})$
x_2	$u_2 = f(\underbrace{w_{02}^{(1)} + w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2}_{z_2^{(1)}})$	

Here, $f(x)$ denotes the activation function, for example, ReLU.

Example of Forward Propagation/Backpropagation

Let's consider the following Neural Network:

input layer	hidden layer	output layer
x_1	$u_1 = f(\underbrace{w_{01}^{(1)} + w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2}_{z_1^{(1)}})$	$\hat{y} = f(\underbrace{w_0^{(2)} + w_1^{(2)} u_1 + w_2^{(2)} u_2}_{z^{(2)}})$
x_2	$u_2 = f(\underbrace{w_{02}^{(1)} + w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2}_{z_2^{(1)}})$	

Here, $f(x)$ denotes the activation function, for example, ReLU.

Forward Propagation: Given weights $\boldsymbol{w}$ and inputs x_1, x_2, compute

- $z_1^{(1)}$ and $z_2^{(1)}$
- u_1 and u_2
- $\hat{y}$

Example of Forward Propagation/Backpropagation

Backpropagation:

Given weights $\boldsymbol{w}$, inputs x_1, x_2, and $z_1^{(1)}$, $z_2^{(1)}$, u_1, u_2, $\hat{y}$, compute

- Error associated with the output layer:

$$\varepsilon^{(2)} \doteq \frac{\partial L}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}} \left[(\hat{y} - y)^2 \right] = 2(\hat{y} - y)$$

- Errors associated with the hidden layer:

$$\varepsilon_h^{(1)} \doteq \frac{\partial L}{\partial u_h} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u_h} = \varepsilon^{(2)} f'(z^{(2)}) w_h^{(2)}, \quad h = 1, 2.$$

Example of Forward Propagation/Backpropagation

Computation of $\nabla L(\boldsymbol{w})$:

- Partial derivatives of the loss function with respect to weights in the output layer:

$$\frac{\partial L}{\partial w_h^{(2)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_h^{(2)}} = \varepsilon^{(2)} \frac{\partial}{\partial w_h^{(2)}} \left[f(\underbrace{w_0^{(2)} + w_1^{(2)} u_1 + w_2^{(2)} u_2}_{z^{(2)}}) \right] = \varepsilon^{(2)} f'(z^{(2)}) u_h,$$

  where $h = 0, 1, 2$.

- Partial derivatives of the loss function with respect to weights in the hidden layer:

$$\frac{\partial L}{\partial w_{jh}^{(1)}} = \frac{\partial L}{\partial u_h} \frac{\partial u_h}{\partial w_{jh}^{(1)}} = \varepsilon_h^{(1)} \frac{\partial}{\partial w_{jh}^{(1)}} \left[f(\underbrace{w_{0h}^{(1)} + w_{1h}^{(1)} x_1 + w_{2h}^{(1)} x_2}_{z_h{}^{(1)}}) \right] = \varepsilon_h^{(1)} f'(z_h{}^{(1)}) x_j,$$

  for each $j = 0, 1, 2$ and $h = 1, 2$. Here, we define $x_0 \doteq 1$.

Example of Forward Propagation/Backpropagation

The Stochastic Gradient Descent (SGD) update of the weights using learning rate α:

$$\mathbf{w} := \mathbf{w} - \alpha \nabla L,$$

where $\nabla L \doteq \Big(\underbrace{\frac{\partial L}{\partial w_{01}^{(1)}}, \frac{\partial L}{\partial w_{11}^{(1)}}, \frac{\partial L}{\partial w_{21}^{(1)}}, \frac{\partial L}{\partial w_{02}^{(1)}}, \frac{\partial L}{\partial w_{12}^{(1)}}, \frac{\partial L}{\partial w_{22}^{(1)}}}_{\text{hidden layer}}, \underbrace{\frac{\partial L}{\partial w_{0}^{(2)}}, \frac{\partial L}{\partial w_{1}^{(2)}}, \frac{\partial L}{\partial w_{2}^{(2)}}}_{\text{output layer}} \Big)^T.$

Therefore,

$$\begin{aligned}
\mathbf{w} &:= \mathbf{w} - \alpha \nabla L \\
&= \big(\underbrace{w_{01}^{(1)}, w_{11}^{(1)}, w_{21}^{(1)}, w_{02}^{(1)}, w_{12}^{(1)}, w_{22}^{(1)}}_{\text{hidden layer}}, \underbrace{w_{0}^{(2)}, w_{1}^{(2)}, w_{2}^{(2)}}_{\text{output layer}} \big)^T \\
&- \alpha \Big(\underbrace{\frac{\partial L}{\partial w_{01}^{(1)}}, \frac{\partial L}{\partial w_{11}^{(1)}}, \frac{\partial L}{\partial w_{21}^{(1)}}, \frac{\partial L}{\partial w_{02}^{(1)}}, \frac{\partial L}{\partial w_{12}^{(1)}}, \frac{\partial L}{\partial w_{22}^{(1)}}}_{\text{hidden layer}}, \underbrace{\frac{\partial L}{\partial w_{0}^{(2)}}, \frac{\partial L}{\partial w_{1}^{(2)}}, \frac{\partial L}{\partial w_{2}^{(2)}}}_{\text{output layer}} \Big)^T
\end{aligned}$$

Contents

1. Recurrent Neural Networks (RNN)
   - Recurrent Neuron and Layer of Recurrent Neurons
   - Memory Cells
   - Long Short-Term Memory (LSTM) Cell
   - Gated Recurrent Unit (GRU) Cell
2. Unstable Gradients
   - Vanishing/Exploding Gradients Problems
   - Techniques to Alleviate the Unstable Gradient Problems
3. Neural Network Optimization Algorithms
   - Objective (Cost) Function
   - SGD, mini-batch GD, and GD Optimization
   - Forward Propagation and Backpropagation
   - Momentum Optimization, NAG, AdaGrad, RMSProp, and Adam

Momentum Optimization

The Momentum Optimization algorithm is defined as follows:
Fist, *momentum vector* $\mathbf{v}$ is initialized at $\mathbf{0}$ and then the updates are

$$\mathbf{v} := -\alpha \underbrace{\frac{1}{s}\sum_{i=1}^{s} \nabla L^{(i)}(\boldsymbol{w})}_{\approx \nabla J(\boldsymbol{w})} + \eta\, \mathbf{v}$$

$$\boldsymbol{w} := \boldsymbol{w} + \mathbf{v}$$

where $L^{(i)}(\boldsymbol{w})$ is based on one observation i and $1 \leq s \leq m$, where m denotes the total number of observations in the data set.

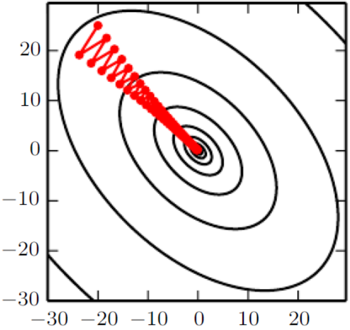
The hyperparameters of the algorithm are

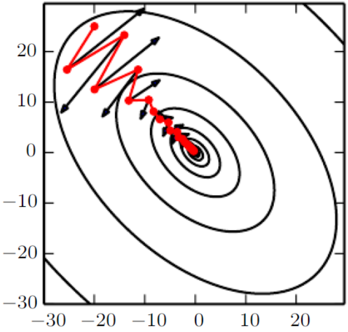
- s - mini-batch size
- α - learning rate
- η - *momentum*, a number between 0 and 1

Momentum Optimization

Example: Path in (w_1, w_2) plane.

Left: no momentum, i.e. $\eta = 0$. Right: Momentum optimization with $\eta > 0$.





Source: *Deep Learning* by I. Goodfellow, Y. Bengio, and A. Courville

Momentum Optimization

Example: Momentum Optimization with $s = 128$, $\alpha = 0.05$, and $\eta = 0.9$.

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 35
model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
              optimizer=keras.optimizers.SGD(lr=0.05, momentum=0.9))

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

Nesterov Accelerated Gradient (NAG)

The Nesterov Accelerated Gradient (NAG) algorithm is defined as follows: Fist, *momentum vector* $\mathbf{v}$ is initialized at $\mathbf{0}$ and then the updates are

$$\mathbf{v} := -\,\alpha \underbrace{\frac{1}{s}\sum_{i=1}^{s} \nabla L^{(i)}(\boldsymbol{w} + \eta\,\mathbf{v})}_{\approx \nabla J(\boldsymbol{w})} + \eta\,\mathbf{v}$$

$$\boldsymbol{w} := \boldsymbol{w} + \mathbf{v}$$

where $L^{(i)}(\boldsymbol{w} + \eta\,\mathbf{v})$ is based on one observation i and $1 \leq s \leq m$, where m denotes the total number of observations in the data set.

The hyperparameters of the algorithm are

- s - mini-batch size
- α - learning rate
- η - *momentum*, a number between 0 and 1

Nesterov Accelerated Gradient (NAG)

Example: Nesterov Accelerated Gradient (NAG) with $s = 128$, $\alpha = 0.05$, and $\eta = 0.9$.

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_35 (Dense)             (None, 512)               401920    
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656    
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130      
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 35
model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
              optimizer=keras.optimizers.SGD(lr=0.05, momentum=0.9, nesterov=True))

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

AdaGrad

The AdaGrad algorithm is defined as follows:
Fist, initialize vector $\mathbf{r}$ (with $r_k > 0$) and then the updates are

$$\mathbf{g} := \underbrace{\frac{1}{s}\sum_{i=1}^{s} \nabla L^{(i)}(\boldsymbol{w})}_{\approx \nabla J(\boldsymbol{w})}$$

$$\mathbf{r} := \mathbf{r} + \mathbf{g} \odot \mathbf{g}$$

$$\boldsymbol{w} := \boldsymbol{w} - \frac{\alpha}{\sqrt{\mathbf{r} + \epsilon}} \odot \mathbf{g}$$

where $L^{(i)}(\boldsymbol{w})$ is based on one observation i and $1 \leq s \leq m$, where m denotes the total number of observations in the data set. $\odot$ denotes element-wise multiplication. The hyperparameters of the algorithm are

- s - mini-batch size
- α - learning rate
- ϵ - positive small parameter, typically around 10^{-7}

AdaGrad

Example: AdaGrad with $s = 128$, $\alpha = 0.05$, $\epsilon = 10^{-5}$, and r_k initialized at 0.1.

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 3
model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
              optimizer=keras.optimizers.Adagrad(lr=0.05, epsilon=1e-5))

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

RMSProp

The RMSProp is defined as follows:
Fist, initialize vector $\mathbf{r}$ (with $r_k > 0$) and then the updates are

$$\mathbf{g} := \underbrace{\frac{1}{s}\sum_{i=1}^{s} \nabla L^{(i)}(\boldsymbol{w})}_{\approx \nabla J(\boldsymbol{w})}$$

$$\mathbf{r} := \rho\, \mathbf{r} + (1-\rho)\mathbf{g} \odot \mathbf{g}$$

$$\boldsymbol{w} := \boldsymbol{w} - \frac{\alpha}{\sqrt{\mathbf{r} + \epsilon}} \odot \mathbf{g}$$

where $L^{(i)}(\boldsymbol{w})$ is based on one observation i and $1 \leq s \leq m$, where m denotes the total number of observations in the data set. $\odot$ denotes element-wise multiplication. The hyperparameters of the algorithm are

- s - mini-batch size
- α - learning rate
- ϵ - positive small parameter, typically around 10^{-7}
- ρ - decay rate between 0 and 1, typically around 0.9

RMSProp

Example: AdaGrad with $s = 128$, $\alpha = 0.05$, $\epsilon = 10^{-5}$, and $\rho = 0.9$.

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 35
model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
              optimizer=keras.optimizers.RMSprop(lr=0.05, rho=0.9, epsilon=1e-07))

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

Adam

Let's combine the Momentum Optimization and RMSProp:

Momentum Optimization	RMSProp
$\mathbf{g} := \underbrace{\frac{1}{s}\sum_{i=1}^{s} \nabla L^{(i)}(\boldsymbol{w})}_{\approx \nabla J(\boldsymbol{w})}$ $\mathbf{v} := \mathbf{g} + \eta\, \boldsymbol{v}$ $\boldsymbol{w} := \boldsymbol{w} - \alpha \mathbf{v}$	$\mathbf{g} := \underbrace{\frac{1}{s}\sum_{i=1}^{s} \nabla L^{(i)}(\boldsymbol{w})}_{\approx \nabla J(\boldsymbol{w})}$ $\mathbf{r} := \rho\, \mathbf{r} + (1-\rho)\mathbf{g} \odot \mathbf{g}$ $\boldsymbol{w} := \boldsymbol{w} - \frac{\alpha}{\sqrt{\mathbf{r} + \epsilon}} \odot \mathbf{g}$

to get Adam (Adaptive Momentum):

Adam

Fist, initialize *momentum vector* $\mathbf{v} = \mathbf{0}$ and vector $\mathbf{r}$ (with $r_k > 0$), then the updates at iteration step t are

$$\begin{aligned}
\mathbf{g} &:= \underbrace{\frac{1}{s}\sum_{i=1}^{s} \nabla L^{(i)}(\boldsymbol{w})}_{\approx \nabla J(\boldsymbol{w})}, \\
\mathbf{v} &:= (1-\beta_1)\mathbf{g} + \beta_1 \mathbf{v}, \quad \mathbf{v} := \frac{\mathbf{v}}{1-\beta_1^t}, \\
\mathbf{r} &:= \beta_2\, \mathbf{r} + (1-\beta_2)\mathbf{g} \odot \mathbf{g}, \quad \mathbf{r} := \frac{\mathbf{r}}{1-\beta_2^t}, \\
\boldsymbol{w} &:= \boldsymbol{w} - \frac{\alpha}{\sqrt{\mathbf{r}+\epsilon}} \odot \mathbf{v},
\end{aligned}$$

where $L^{(i)}(\boldsymbol{w})$ is based on one observation i and $1 \leq s \leq m$, where m denotes the total number of observations in the data set. $\odot$ denotes element-wise multiplication. The hyperparameters of the algorithm are

- s is the mini-batch size and α is learning rate
- β_1 - *momentum*, a number between 0 and 1 (analogous to η in Momentum Opt.)
- β_2 - decay rate between 0 and 1, typically around 0.9 (analogous to ρ in RMSProp)
- ϵ - positive small parameter, typically around 10^{-7}

Example: Adam with $s = 128$, $\alpha = 0.001$, $\epsilon = 10^{-7}$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 35
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='adam')

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

Example: Adam with $s = 128$, $\alpha = 0.05$, $\epsilon = 10^{-5}$, $\beta_1 = 0.85$, and $\beta_2 = 0.95$

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 35

model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
              optimizer=keras.optimizers.adam(lr=0.05, beta_1=0.85, beta_2=0.95, epsilon=1e-05))

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```