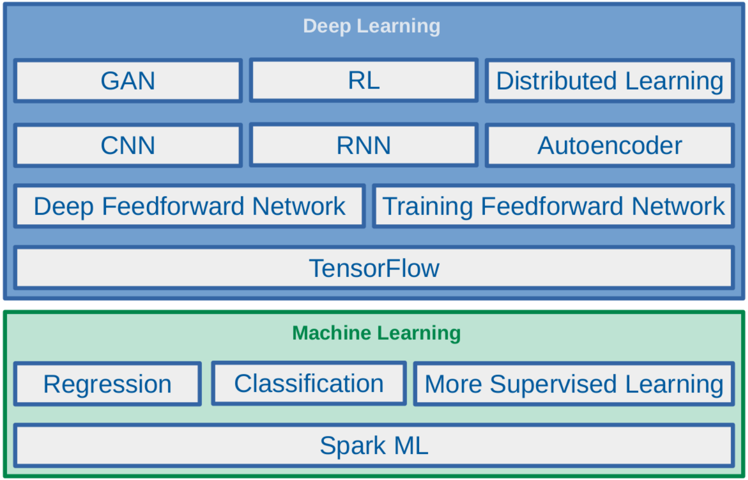
Training Deep Feedforwards Networks

Amir H. Payberah
payberah@kth.se
19/11/2019

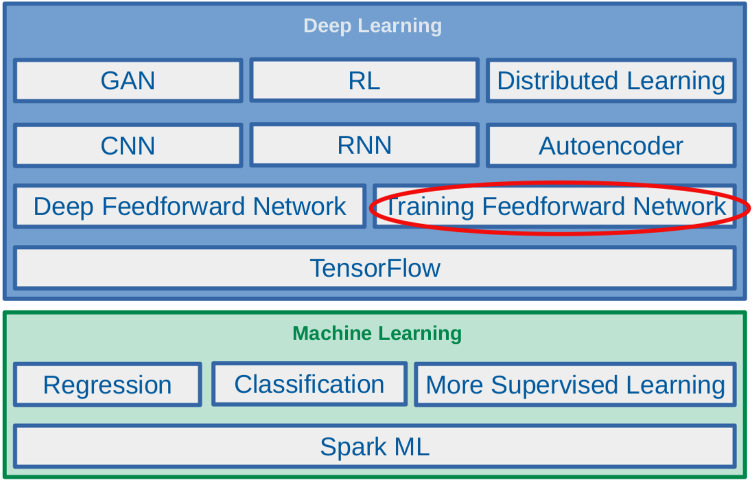
The Course Web Page

https://id2223kth.github.io

Where Are We?

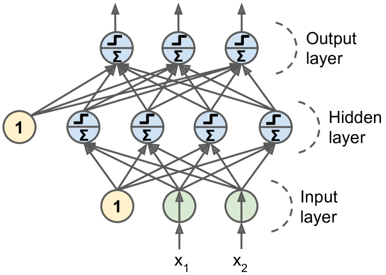

Deep Learning
GAN
RL
Distributed Learning
CNN
RNN
Autoencoder
Deep Feedforward Network
Training Feedforward Network
TensorFlow
Machine Learning
Regression
Classification
More Supervised Learning
Spark ML

Where Are We?

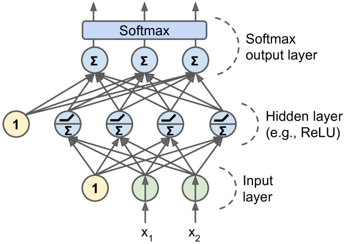

Deep Learning
GAN
RL
Distributed Learning
CNN
RNN
Autoencoder
Deep Feedforward Network
Training Feedforward Network
TensorFlow
Machine Learning
Regression
Classification
More Supervised Learning
Spark ML

Feedforward Neural Network Architecture

- A feedforward neural network is composed of:
  - One input layer
  - One or more hidden layers
  - One final output layer



Feedforward Network in TensorFlow




```python
n_output = 3
n_hidden = 4
n_features = 2

model = keras.models.Sequential()
model.add(keras.layers.Dense(n_hidden, input_shape=(n_features,), activation="relu"))
model.add(keras.layers.Dense(n_output, activation="softmax"))

model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd", metrics=["accuracy"])
model.fit(X_train, y_train, epochs=30)
```

Challenges of Training Feedforward Neural Networks

- Challenges ...

Challenges of Training Feedforward Neural Networks

- Challenges ...
- Overfitting: risk of overfitting a model with large number of parameters.

Challenges of Training Feedforward Neural Networks

- Challenges ...
- Overfitting: risk of overfitting a model with large number of parameters.
- Vanishing/exploding gradients: hard to train lower layers.

Challenges of Training Feedforward Neural Networks

- Challenges ...
- Overfitting: risk of overfitting a model with large number of parameters.
- Vanishing/exploding gradients: hard to train lower layers.
- Training speed: slow training with large networks.

Overfitting

High Degree of Freedom and Overfitting Problem

- With large number of parameters, a network has a high degree of freedom.
- It can fit a huge variety of complex datasets.

High Degree of Freedom and Overfitting Problem

- With large number of parameters, a network has a high degree of freedom.
- It can fit a huge variety of complex datasets.
- This flexibility also means that it is prone to overfitting on training set.

High Degree of Freedom and Overfitting Problem

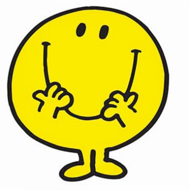
- With large number of parameters, a network has a high degree of freedom.
- It can fit a huge variety of complex datasets.
- This flexibility also means that it is prone to overfitting on training set.
- Let's reduce the degree of freedom a model.

Avoiding Overfitting

- Early stopping
- $l1$ and $l2$ regularization
- Max-norm regularization
- Dropout
- Data augmentation

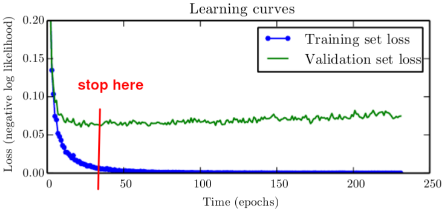
Avoiding Overfitting

- Early stopping
- $l1$ and $l2$ regularization
- Max-norm regularization
- Dropout
- Data augmentation



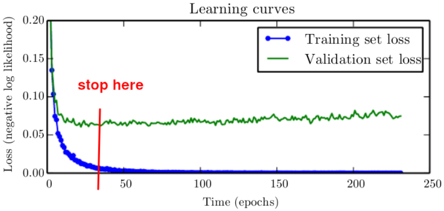
Early Stopping (1/2)

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



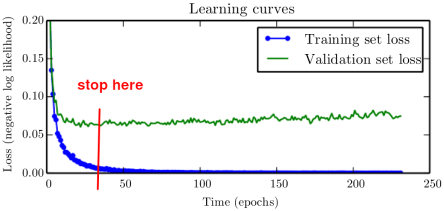
Early Stopping (1/2)

- 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 (1/2)

- 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.

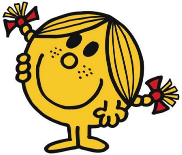


Early Stopping (2/2)

```
from tensorflow.keras.callbacks import EarlyStopping

model = tf.keras.models.Sequential(...)

model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

earlystop_callback = EarlyStopping(monitor='accuracy', min_delta=0.05, patience=1)

model.fit(x_train, y_train, epochs=500, callbacks=[earlystop_callback])
```

Avoiding Overfitting

- Early stopping
- $l1$ and $l2$ regularization
- Max-norm regularization
- Dropout
- Data augmentation



$l1$ and $l2$ Regularization (1/3)

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

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

- Two questions:
  1. How should we define $\mathtt{R}(\mathtt{w})$?
  2. How do we determine λ?

$l1$ and $l2$ Regularization (2/3)

- $l1$ regression: $\mathtt{R}(\mathbf{w}) = \lambda \sum_{\mathtt{i}=1}^{\mathtt{n}} |\mathtt{w_i}|$ is added to the cost function.

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

```
keras.layers.Dense(100, activation="relu", kernel_regularizer=keras.regularizers.l1(0.1))
```

$l1$ and $l2$ Regularization (3/3)

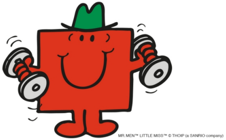
- $l2$ regression: $R(\mathbf{w}) = \lambda \sum_{i=1}^{n} w_i^2$ is added to the cost function.

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \lambda \sum_{i=1}^{n} w_i^2$$

```
keras.layers.Dense(100, activation="relu", kernel_regularizer=keras.regularizers.l2(0.01))
```

Avoiding Overfitting

- Early stopping
- $l1$ and $l2$ regularization
- Max-norm regularization
- Dropout
- Data augmentation



Max-Norm Regularization

- Max-norm regularization: constrains the weights $\mathbf{w}_j$ of the incoming connections for each neuron j.
  - Prevents them from getting too large.

Max-Norm Regularization

- Max-norm regularization: constrains the weights $\mathbf{w}_j$ of the incoming connections for each neuron j.
  - Prevents them from getting too large.

- After each training step, clip $\mathbf{w}_j$ as below, if $||\mathbf{w}_j||_2 > r$:

$$\mathbf{w}_j \leftarrow \mathbf{w}_j \frac{r}{||\mathbf{w}_j||_2}$$

  - r is the max-norm hyperparameter
  - $||\mathbf{w}_j||_2 = (\sum_i w_{i,j}^2)^{\frac{1}{2}} = \sqrt{w_{1,j}^2 + w_{2,j}^2 + \cdots + w_{n,j}^2}$

Max-Norm Regularization

- Max-norm regularization: constrains the weights $\mathbf{w}_j$ of the incoming connections for each neuron j.
  - Prevents them from getting too large.

- After each training step, clip $\mathbf{w}_j$ as below, if $||\mathbf{w}_j||_2 > r$:

$$\mathbf{w}_j \leftarrow \mathbf{w}_j \frac{r}{||\mathbf{w}_j||_2}$$

  - r is the max-norm hyperparameter
  - $||\mathbf{w}_j||_2 = (\sum_i w_{i,j}^2)^{\frac{1}{2}} = \sqrt{w_{1,j}^2 + w_{2,j}^2 + \cdots + w_{n,j}^2}$

```
keras.layers.Dense(100, activation="relu", kernel_constraint=keras.constraints.max_norm(1.))
```

Avoiding Overfitting

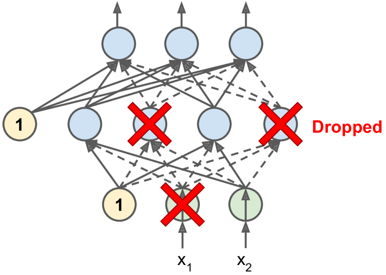
- Early stopping
- $l1$ and $l2$ regularization
- Max-norm regularization
- Dropout
- Data augmentation

Dropout (1/4)

- Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work?

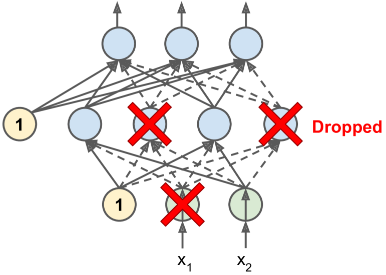
Dropout (2/4)

- At each training step, each neuron drops out temporarily with a probability p.



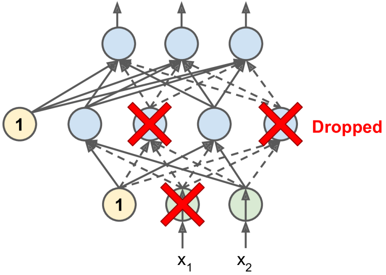
Dropout (2/4)

- At each training step, each neuron drops out temporarily with a probability p.
  - The hyperparameter p is called the dropout rate.



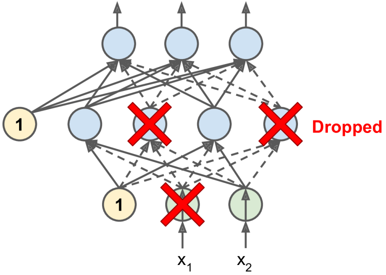
Dropout (2/4)

- 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.



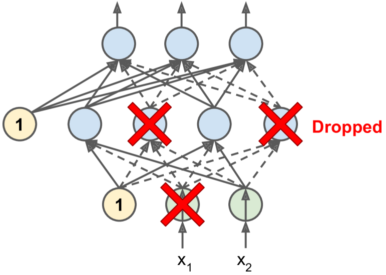
Dropout (2/4)

- 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.



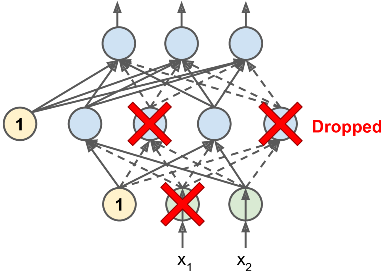
Dropout (2/4)

- 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.



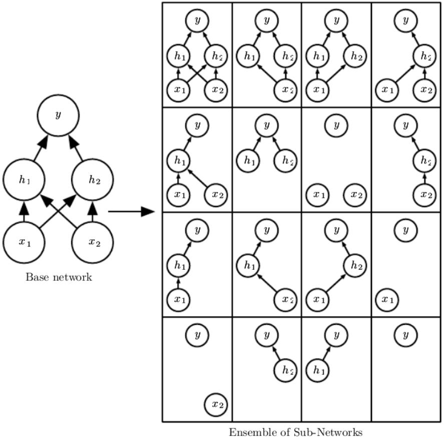
Dropout (2/4)

- 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.



Dropout (3/4)

- 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.



Dropout (4/4)

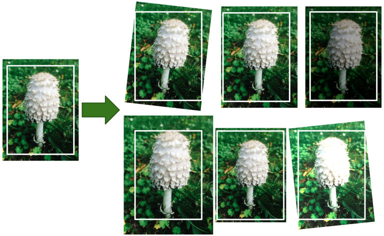
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

Avoiding Overfitting

- Early stopping
- $l1$ and $l2$ regularization
- Max-norm regularization
- Dropout
- Data augmentation

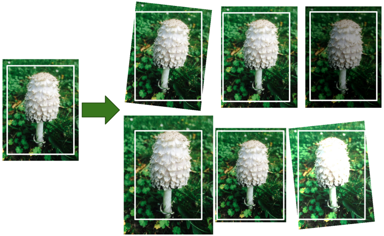
Data Augmentation

- 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.
  - E.g., in an image classification we can slightly shift, rotate and resize an image.
  - Add the resulting pictures to the training set.



Vanishing/Exploding Gradients

Vanishing/Exploding Gradients Problem (1/4)

- The backpropagation goes from output to input layer, and propagates the error gradient on the way.

$$\mathtt{w}^{(\mathtt{next})} = \mathtt{w} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}}$$

Vanishing/Exploding Gradients Problem (1/4)

- The backpropagation goes from output to input layer, and propagates the error gradient on the way.

$$\mathtt{w}^{(\mathtt{next})} = \mathtt{w} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}}$$

- Gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
- As a result, the gradient descent update leaves the lower layer connection weights virtually unchanged.

Vanishing/Exploding Gradients Problem (1/4)

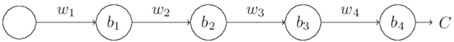
- The backpropagation goes from output to input layer, and propagates the error gradient on the way.

$$\mathtt{w}^{(\mathtt{next})} = \mathtt{w} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}}$$

- Gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
- As a result, the gradient descent update leaves the lower layer connection weights virtually unchanged.
- This is called the vanishing gradients problem.

Vanishing/Exploding Gradients Problem (2/4)

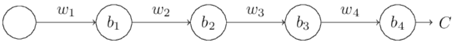
- Assume a network with just a single neuron in each layer.



  - $w_1, w_2, \cdots$ are the weights
  - $b_1, b_2, \cdots$ are the biases
  - C is the cost function

Vanishing/Exploding Gradients Problem (2/4)

- Assume a network with just a single neuron in each layer.

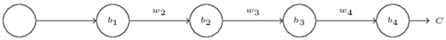


  - $w_1, w_2, \cdots$ are the weights
  - $b_1, b_2, \cdots$ are the biases
  - C is the cost function

- The output a_j from the jth neuron is $\sigma(z_j)$.
  - σ is the sigmoid activation function
  - $z_j = w_j a_{j-1} + b_j$
  - E.g., $a_4 = \sigma(z_4) = \mathtt{sigmoid}(w_4 a_3 + b_4)$

Vanishing/Exploding Gradients Problem (3/4)

- Lets compute the gradient associated to the first hidden neuron ($\frac{\partial C}{\partial b_1}$).



$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial z_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial z_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial z_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial z_1}{\partial b_1}$$

Vanishing/Exploding Gradients Problem (3/4)

- Lets compute the gradient associated to the first hidden neuron ($\frac{\partial C}{\partial b_1}$).

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial z_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial z_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial z_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial z_1}{\partial b_1}$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial w_4 a_3 + b_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial w_3 a_2 + b_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial w_2 a_1 + b_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial w_1 a_0 + b_1}{\partial b_1}$$

Vanishing/Exploding Gradients Problem (3/4)

- Lets compute the gradient associated to the first hidden neuron ($\frac{\partial C}{\partial b_1}$).

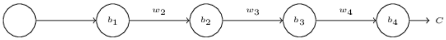
$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial z_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial z_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial z_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial z_1}{\partial b_1}$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial w_4 a_3 + b_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial w_3 a_2 + b_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial w_2 a_1 + b_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial w_1 a_0 + b_1}{\partial b_1}$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times w_4 \times \frac{\partial a_3}{\partial z_3} \times w_3 \times \frac{\partial a_2}{\partial z_2} \times \times w_2 \times \frac{\partial a_1}{\partial z_1} \times 1$$

Vanishing/Exploding Gradients Problem (4/4)

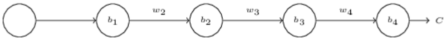
- Now, consider $\frac{\partial C}{\partial b_3}$.



$$\frac{\partial C}{\partial b_3} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times w_4 \times \frac{\partial a_3}{\partial z_3}$$

Vanishing/Exploding Gradients Problem (4/4)

- Now, consider $\frac{\partial C}{\partial b_3}$.

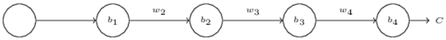


$$\frac{\partial C}{\partial b_3} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times w_4 \times \frac{\partial a_3}{\partial z_3}$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times w_4 \times \frac{\partial a_3}{\partial z_3} \times w_3 \times \frac{\partial a_2}{\partial z_2} \times w_2 \times \frac{\partial a_1}{\partial z_1} \times 1$$

Vanishing/Exploding Gradients Problem (4/4)

- Now, consider $\frac{\partial C}{\partial b_3}$.



$$\frac{\partial C}{\partial b_3} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times w_4 \times \frac{\partial a_3}{\partial z_3}$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times w_4 \times \frac{\partial a_3}{\partial z_3} \times w_3 \times \frac{\partial a_2}{\partial z_2} \times w_2 \times \frac{\partial a_1}{\partial z_1} \times 1$$

- Assume $w_3 \times \frac{\partial a_2}{\partial z_2} < \frac{1}{4}$ and $w_2 \times \frac{\partial a_1}{\partial z_1} < \frac{1}{4}$
  - The gradient $\frac{\partial C}{\partial b_1}$ be a factor of 16 (or more) smaller than $\frac{\partial C}{\partial b_3}$.
  - This is the essential origin of the vanishing gradient problem.

Overcoming the Vanishing Gradient

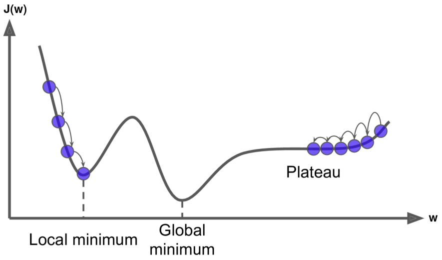
- Parameter initialization strategies
- Nonsaturating activation function
- Batch normalization
- Gradient clipping

Overcoming the Vanishing Gradient

- Parameter initiazlization strategies
- Nonsaturating activation function
- Batch normalization
- Gradient clipping

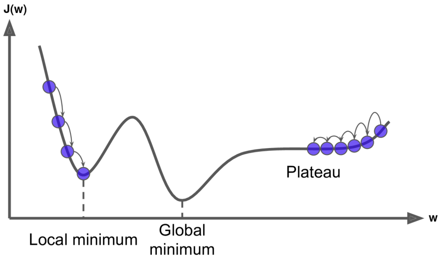
Parameter Initialization Strategies (1/4)

- The non-linearity of a neural network causes the cost functions to become non-convex.



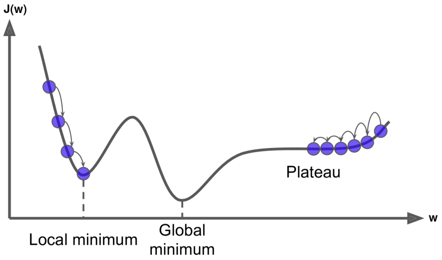
Parameter Initialization Strategies (1/4)

- The non-linearity of a neural network causes the cost functions to become non-convex.
- The stochastic gradient descent on non-convex cost functions performs is sensitive to the values of the initial parameters.



Parameter Initialization Strategies (1/4)

- The non-linearity of a neural network causes the cost functions to become non-convex.
- The stochastic gradient descent on non-convex cost functions performs is sensitive to the values of the initial parameters.
- Designing initialization strategies is a difficult task.



Parameter Initialization Strategies (2/4)

- The initial parameters need to break symmetry between different units.

Parameter Initialization Strategies (2/4)

- The initial parameters need to break symmetry between different units.
- Two hidden units with the same activation function connected to the same inputs, must have different initial parameters.

Parameter Initialization Strategies (2/4)

- The initial parameters need to break symmetry between different units.
- Two hidden units with the same activation function connected to the same inputs, must have different initial parameters.
  - The goal of having each unit compute a different function.

Parameter Initialization Strategies (2/4)

- The initial parameters need to break symmetry between different units.
- Two hidden units with the same activation function connected to the same inputs, must have different initial parameters.
  - The goal of having each unit compute a different function.
- It motivates random initialization of the parameters.
  - Typically, we set the biases to constants, and initialize only the weights randomly.

Parameter Initialization Strategies (3/4)

- We need the signals to flow properly in both directions.

Parameter Initialization Strategies (3/4)

- We need the signals to flow properly in both directions.
- The Glorot and Bengio initialization proposed that:

Parameter Initialization Strategies (3/4)

- We need the signals to flow properly in both directions.
- The Glorot and Bengio initialization proposed that:
  - The variance of the outputs of each layer to be equal to the variance of its inputs.

Parameter Initialization Strategies (3/4)

- We need the signals to flow properly in both directions.
- The Glorot and Bengio initialization proposed that:
  - The variance of the outputs of each layer to be equal to the variance of its inputs.
  - The gradients to have equal variance before and after flowing through a layer in the reverse direction.

Parameter Initialization Strategies (3/4)

- We need the signals to flow properly in both directions.
- The Glorot and Bengio initialization proposed that:
  - The variance of the outputs of each layer to be equal to the variance of its inputs.
  - The gradients to have equal variance before and after flowing through a layer in the reverse direction.
- It is not possible to guarantee both unless each layer has an equal number of inputs and neurons.

Parameter Initialization Strategies (3/4)

- We need the signals to flow properly in both directions.
- The Glorot and Bengio initialization proposed that:
  - The variance of the outputs of each layer to be equal to the variance of its inputs.
  - The gradients to have equal variance before and after flowing through a layer in the reverse direction.
- It is not possible to guarantee both unless each layer has an equal number of inputs and neurons.
- Based on the Xavier initialization, the weights are initialized using normal distribution with mean 0 and the following standard deviation.

Parameter Initialization Strategies (4/4)

- $\texttt{fan}_{\texttt{in}}$ and $\texttt{fan}_{\texttt{out}}$ are the number of inputs and neurons for the layer whose weights are being initialized.
- $\texttt{fan}_{\texttt{avg}} = \frac{2}{\texttt{fan}_{\texttt{in}} + \texttt{fan}_{\texttt{out}}}$

Parameter Initialization Strategies (4/4)

- $\texttt{fan}_{\texttt{in}}$ and $\texttt{fan}_{\texttt{out}}$ are the number of inputs and neurons for the layer whose weights are being initialized.
- $\texttt{fan}_{\texttt{avg}} = \frac{2}{\texttt{fan}_{\texttt{in}} + \texttt{fan}_{\texttt{out}}}$
- Glorot initialization, for none, logistic, sigmoid, and tanh: $\sigma^2 = \frac{1}{\texttt{fan}_{\texttt{avg}}}$

Parameter Initialization Strategies (4/4)

- $\texttt{fan}_{\texttt{in}}$ and $\texttt{fan}_{\texttt{out}}$ are the number of inputs and neurons for the layer whose weights are being initialized.
- $\texttt{fan}_{\texttt{avg}} = \frac{2}{\texttt{fan}_{\texttt{in}} + \texttt{fan}_{\texttt{out}}}$
- Glorot initialization, for none, logistic, sigmoid, and tanh: $\sigma^2 = \frac{1}{\texttt{fan}_{\texttt{avg}}}$
- He initialization, for ReLU: $\sigma^2 = \frac{2}{\texttt{fan}_{\texttt{in}}}$

Parameter Initialization Strategies (4/4)

- $\mathtt{fan}_{\mathtt{in}}$ and $\mathtt{fan}_{\mathtt{out}}$ are the number of inputs and neurons for the layer whose weights are being initialized.
- $\mathtt{fan}_{\mathtt{avg}} = \frac{2}{\mathtt{fan}_{\mathtt{in}} + \mathtt{fan}_{\mathtt{out}}}$
- Glorot initialization, for none, logistic, sigmoid, and tanh: $\sigma^2 = \frac{1}{\mathtt{fan}_{\mathtt{avg}}}$
- He initialization, for ReLU: $\sigma^2 = \frac{2}{\mathtt{fan}_{\mathtt{in}}}$

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

Overcoming the Vanishing Gradient

- Parameter initiazlization strategies
- Nonsaturating activation function
- Batch normalization
- Gradient clipping

Nonsaturating Activation Functions (1/4)

- $\texttt{ReLU}(z) = \max(0, z)$
- The dying ReLUs problem.

Nonsaturating Activation Functions (1/4)

- $\texttt{ReLU}(z) = \max(0, z)$

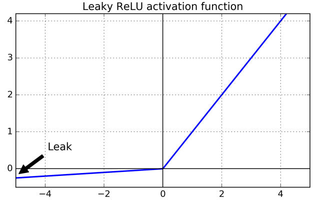
- The dying ReLUs problem.
  - During training, some neurons stop outputting anything other than 0.
  - E.g., when the weighted sum of the neuron's inputs is negative, it starts outputting 0.

Nonsaturating Activation Functions (1/4)

- $\texttt{ReLU}(z) = \max(0, z)$

- The dying ReLUs problem.
  - During training, some neurons stop outputting anything other than 0.
  - E.g., when the weighted sum of the neuron's inputs is negative, it starts outputting 0.

- Use leaky ReLU instead: $\texttt{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$.
  - α is the slope of the function for $z < 0$.



Nonsaturating Activation Functions (2/4)

- Randomized Leaky ReLU (RReLU)
  - α is picked randomly during training, and it is fixed during testing.

Nonsaturating Activation Functions (2/4)

- Randomized Leaky ReLU (RReLU)
  - α is picked randomly during training, and it is fixed during testing.
- Parametric Leaky ReLU (PReLU)
  - Learn α during training (instead of being a hyperparameter).

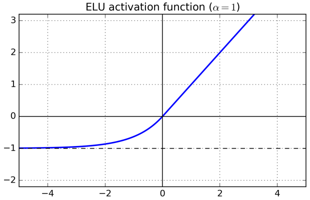
Nonsaturating Activation Functions (2/4)

- Randomized Leaky ReLU (RReLU)
  - α is picked randomly during training, and it is fixed during testing.

- Parametric Leaky ReLU (PReLU)
  - Learn α during training (instead of being a hyperparameter).

- Exponential Linear Unit (ELU)

$$\mathrm{ELU}_\alpha(\mathbf{z}) = \begin{cases} \alpha(\exp(\mathbf{z}) - 1) & \text{if} \quad \mathbf{z} < 0 \\ \mathbf{z} & \text{if} \quad \mathbf{z} \geq 0 \end{cases}$$

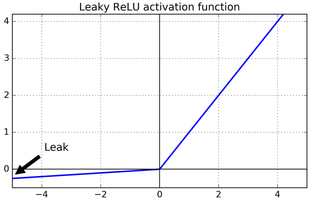


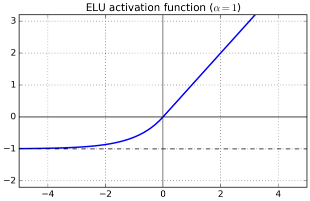
Nonsaturating Activation Functions (3/4)

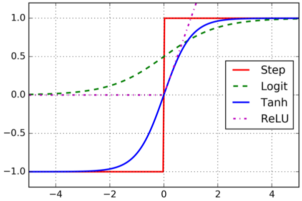
- Which activation function should we use?

Nonsaturating Activation Functions (3/4)

- Which activation function should we use?
- In general logistic $<$ tanh $<$ ReLU $<$ leaky ReLU (and its variants) $<$ ELU

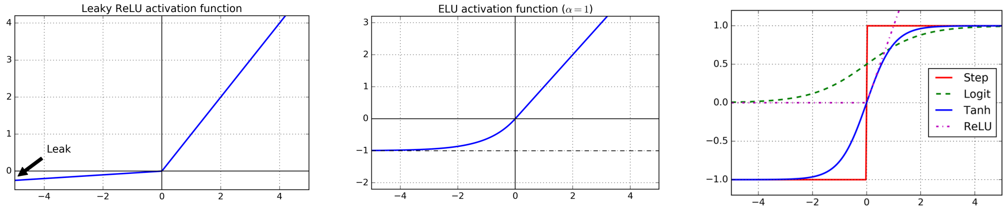






Nonsaturating Activation Functions (3/4)

- Which activation function should we use?
- In general logistic $<$ tanh $<$ ReLU $<$ leaky ReLU (and its variants) $<$ ELU
- If you care about runtime performance, then leaky ReLUs works better than ELUs.



Nonsaturating Activation Functions (4/4)

```
# elu
keras.layers.Dense(10, activation="elu")
```

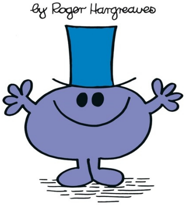
Nonsaturating Activation Functions (4/4)

```
# elu
keras.layers.Dense(10, activation="elu")
```

```
# leaky relu
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(128, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(),
    keras.layers.Dense(10, activation="softmax")
])
```

Overcoming the Vanishing Gradient

- Parameter initiazlization strategies
- Nonsaturating activation function
- Batch normalization
- Gradient clipping



Batch Normalization (1/4)

- The gradient tells how to update each parameter, under the assumption that the other layers do not change.

Batch Normalization (1/4)

- The gradient tells how to update each parameter, under the assumption that the other layers do not change.
  - In practice, we update all of the layers simultaneously.
  - However, unexpected results can happen.

Batch Normalization (1/4)

- The gradient tells how to update each parameter, under the assumption that the other layers do not change.
  - In practice, we update all of the layers simultaneously.
  - However, unexpected results can happen.
- Batch normalization makes the learning of layers in the network more independent of each other.

Batch Normalization (1/4)

- The gradient tells how to update each parameter, under the assumption that the other layers do not change.
  - In practice, we update all of the layers simultaneously.
  - However, unexpected results can happen.
- Batch normalization makes the learning of layers in the network more independent of each other.
  - It is a technique to address the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.

Batch Normalization (1/4)

- The gradient tells how to update each parameter, under the assumption that the other layers do not change.
  - In practice, we update all of the layers simultaneously.
  - However, unexpected results can happen.
- Batch normalization makes the learning of layers in the network more independent of each other.
  - It is a technique to address the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.
- The technique consists of adding an operation in the model just before the activation function of each layer.

Batch Normalization (2/4)

- It's zero-centering and normalizing the inputs, then scaling and shifting the result.

Batch Normalization (2/4)

- It's zero-centering and normalizing the inputs, then scaling and shifting the result.
  - Estimates the inputs' mean and standard deviation of the current mini-batch.

Batch Normalization (2/4)

- It's zero-centering and normalizing the inputs, then scaling and shifting the result.
  - Estimates the inputs' mean and standard deviation of the current mini-batch.

$$\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

$$\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2$$

- μ_B: the empirical mean, evaluated over the whole mini-batch B.
- σ_B: the empirical standard deviation, also evaluated over the whole mini-batch.
- m_B: the number of instances in the mini-batch.

Batch Normalization (3/4)

$$\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\mathbf{z}^{(i)} = \gamma \hat{\mathbf{x}}^{(i)} + \beta$$

- $\hat{\mathbf{x}}^{(i)}$: the zero-centered and normalized input.
- $\mathbf{z}^{(i)}$: the output of the BN operation, which is a scaled and shifted version of the inputs.
- γ: the scaling parameter vector for the layer.
- β: the shifting parameter (offset) vector for the layer.
- ϵ: a tiny number to avoid division by zero.
- $\otimes$: represents the element-wise multiplication.

Batch Normalization (4/4)

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

Overcoming the Vanishing Gradient

- Parameter initiazlization strategies
- Nonsaturating activation function
- Batch normalization
- Gradient clipping

Gradient Clipping

- Gradient clipping: clip the gradients during backpropagation so that they never exceed some threshold.

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

Gradient Clipping

- Gradient clipping: clip the gradients during backpropagation so that they never exceed some threshold.

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

- Setting the `clipvalue` or `clipnorm` argument when creating an optimizer.

Gradient Clipping

- Gradient clipping: clip the gradients during backpropagation so that they never exceed some threshold.

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

- Setting the `clipvalue` or `clipnorm` argument when creating an optimizer.
- `clipvalue=1.0` and `clipnorm=1.0`: values between -1.0 and 1.0.

Gradient Clipping

- Gradient clipping: clip the gradients during backpropagation so that they never exceed some threshold.

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

- Setting the `clipvalue` or `clipnorm` argument when creating an optimizer.
- `clipvalue=1.0` and `clipnorm=1.0`: values between -1.0 and 1.0.
- `clipvalue=1.0`: $[0.9, 100.0] \Rightarrow [0.9, 1.0]$

Gradient Clipping

- Gradient clipping: clip the gradients during backpropagation so that they never exceed some threshold.

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

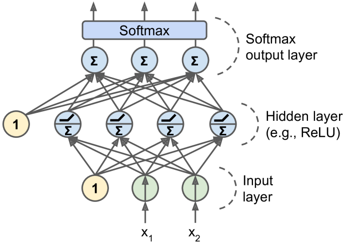
- Setting the `clipvalue` or `clipnorm` argument when creating an optimizer.
- `clipvalue=1.0` and `clipnorm=1.0`: values between -1.0 and 1.0.
- `clipvalue=1.0`: $[0.9, 100.0] \Rightarrow [0.9, 1.0]$
- `clipnorm=1.0`: $[0.9, 100.0] \Rightarrow [0.00899964, 0.9999595]$

Training Speed

Regular Gradient Descent Optimization (1/2)

- Gradient descent optimization algorithm
- It updates the weights $\mathtt{w}_{\mathtt{i}}^{(\mathtt{next})} = \mathtt{w}_{\mathtt{i}} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_{\mathtt{i}}}$
- Better optimization algorithms to improve the training speed

Regular Gradient Descent Optimization (2/2)




```
n_output = 3
n_hidden = 4
n_features = 2

model = keras.models.Sequential()
model.add(keras.layers.Dense(n_hidden, input_shape=(n_features,), activation="relu"))
model.add(keras.layers.Dense(n_output, activation="softmax"))

model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd", metrics=["accuracy"])
model.fit(X_train, y_train, epochs=30)
```

Optimization Algorithms

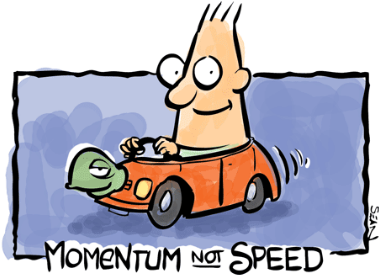
- Momentum
- Nesterov momentum
- AdaGrad
- RMSProp
- Adam Optimization

Optimization Algorithms

- Momentum
- Nesterov momentum
- AdaGrad
- RMSProp
- Adam optimization

Momentum (1/3)

- Momentum is a concept from physics: an object in motion will have a tendency to keep moving.
- It measures the resistance to change in motion.
  - The higher momentum an object has, the harder it is to stop it.

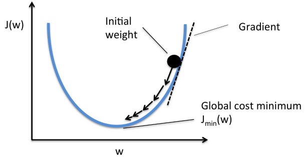


Momentum (2/3)

- This is the very simple idea behind momentum optimization.

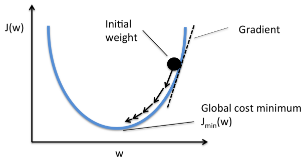
Momentum (2/3)

- This is the very simple idea behind momentum optimization.
- We can see the change in the parameters $\mathbf{w}$ as motion: $\mathtt{w}_{\mathtt{i}}^{(\mathtt{next})} = \mathtt{w}_{\mathtt{i}} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_{\mathtt{i}}}$



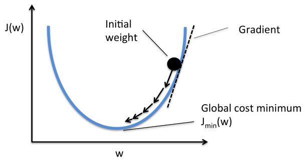
Momentum (2/3)

- This is the very simple idea behind momentum optimization.
- We can see the change in the parameters $\mathbf{w}$ as motion: $\mathtt{w}_\mathtt{i}^{(\mathtt{next})} = \mathtt{w}_\mathtt{i} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_\mathtt{i}}$
- We can thus use the concept of momentum to give the update process a tendency to keep moving in the same direction.



Momentum (2/3)

- This is the very simple idea behind momentum optimization.
- We can see the change in the parameters **w** as motion: $\mathtt{w}_\mathtt{i}^{(\mathtt{next})} = \mathtt{w}_\mathtt{i} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_\mathtt{i}}$
- We can thus use the concept of momentum to give the update process a tendency to keep moving in the same direction.
- It can help to escape from bad local minima pits.



Momentum (3/3)

- Regular gradient descent optimization: $\mathtt{w}_{\mathtt{i}}^{(\mathtt{next})} = \mathtt{w}_{\mathtt{i}} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_{\mathtt{i}}}$

Momentum (3/3)

- Regular gradient descent optimization: $\mathtt{w}_\mathtt{i}^{(\mathtt{next})} = \mathtt{w}_\mathtt{i} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_\mathtt{i}}$
- Momentum optimization cares about what previous gradients were.

Momentum (3/3)

- Regular gradient descent optimization: $\mathtt{w}_\mathtt{i}^{(\mathtt{next})} = \mathtt{w}_\mathtt{i} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_\mathtt{i}}$
- Momentum optimization cares about what previous gradients were.
- At each iteration, it adds the local gradient to the momentum vector **m**.

$$\mathtt{m}_\mathtt{i} = \beta \mathtt{m}_\mathtt{i} + \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_\mathtt{i}}$$

$$\mathtt{w}_\mathtt{i}^{(\mathtt{next})} = \mathtt{w}_\mathtt{i} - \mathtt{m}_\mathtt{i}$$

Momentum (3/3)

- Regular gradient descent optimization: $\mathtt{w}_\mathtt{i}^{(\mathtt{next})} = \mathtt{w}_\mathtt{i} - \eta\frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_\mathtt{i}}$
- Momentum optimization cares about what previous gradients were.
- At each iteration, it adds the local gradient to the momentum vector **m**.

$$\mathtt{m}_\mathtt{i} = \beta\mathtt{m}_\mathtt{i} + \eta\frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_\mathtt{i}}$$

$$\mathtt{w}_\mathtt{i}^{(\mathtt{next})} = \mathtt{w}_\mathtt{i} - \mathtt{m}_\mathtt{i}$$

- β is called momentum, ans it is between 0 and 1.

Momentum (3/3)

- Regular gradient descent optimization: $\mathtt{w}_\mathtt{i}^{(\mathtt{next})} = \mathtt{w}_\mathtt{i} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_\mathtt{i}}$
- Momentum optimization cares about what previous gradients were.
- At each iteration, it adds the local gradient to the momentum vector **m**.

$$\mathtt{m}_\mathtt{i} = \beta \mathtt{m}_\mathtt{i} + \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_\mathtt{i}}$$

$$\mathtt{w}_\mathtt{i}^{(\mathtt{next})} = \mathtt{w}_\mathtt{i} - \mathtt{m}_\mathtt{i}$$

- β is called momentum, ans it is between 0 and 1.

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer, metrics=["accuracy"])
```

Optimization Algorithms

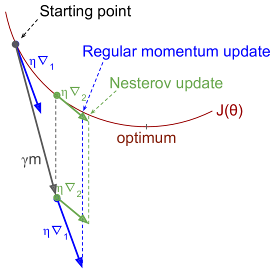
- Momentum
- Nesterov momentum
- AdaGrad
- RMSProp
- Adam optimization

Nesterov Momentum (1/2)

- Nesterov Momentum is a small variant to Momentum optimization.
- Faster than vanilla Momentum optimization.

Nesterov Momentum (1/2)

- Nesterov Momentum is a small variant to Momentum optimization.
- Faster than vanilla Momentum optimization.
- $\nabla 1$ represents the gradient of the cost function measured at the starting point $\mathbf{w}$, and $\nabla 2$ represents the gradient at the point located at $\mathbf{w} + \beta \mathbf{m}$.



Nesterov Momentum (2/2)

- Measure the gradient of the cost function slightly ahead in the direction of the momentum (not at the local position).

$$\mathtt{m_i} = \beta \mathtt{m_i} + \eta \frac{\partial \mathtt{J}(\mathbf{w} + \beta \mathbf{m})}{\partial \mathtt{w_i}}$$

$$\mathtt{w_i^{(next)}} = \mathtt{w_i} - \mathtt{m_i}$$

Nesterov Momentum (2/2)

- Measure the gradient of the cost function slightly ahead in the direction of the momentum (not at the local position).

$$m_i = \beta m_i + \eta \frac{\partial J(\mathbf{w} + \beta \mathbf{m})}{\partial w_i}$$

$$w_i^{(next)} = w_i - m_i$$

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer, metrics=["accuracy"])
```

Optimization Algorithms

- Momentum
- Nesterov momentum
- AdaGrad
- RMSProp
- Adam optimization

AdaGrad (1/2)

- AdaGrad keeps track of a learning rate for each parameter.
- Adapts the learning rate over time (adaptive learning rate).
- Decays the learning rate faster for steep dimensions than for dimensions with gentler slopes.

AdaGrad (2/2)

- For each feature $\mathtt{w_i}$, we do the following steps:

$$\mathtt{s_i} = \mathtt{s_i} + (\frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w_i}})^2$$

$$\mathtt{w_i}^{(\mathtt{next})} = \mathtt{w_i} - \frac{\eta}{\sqrt{\mathtt{s_i} + \epsilon}} \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w_i}}$$

AdaGrad (2/2)

- For each feature $\mathtt{w_i}$, we do the following steps:

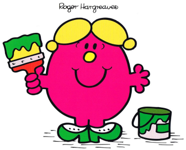
$$\mathtt{s_i} = \mathtt{s_i} + \left(\frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w_i}}\right)^2$$

$$\mathtt{w_i^{(next)}} = \mathtt{w_i} - \frac{\eta}{\sqrt{\mathtt{s_i} + \epsilon}} \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w_i}}$$

```
optimizer = keras.optimizers.Adagrad(lr=0.001)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer, metrics=["accuracy"])
```

Optimization Algorithms

- Momentum
- Nesterov momentum
- AdaGrad
- RMSProp
- Adam optimization



RMSProp (1/2)

- AdaGrad often stops too early when training neural networks.
- The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum.

RMSProp (1/2)

- AdaGrad often stops too early when training neural networks.
- The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum.
- The RMSProp fixed the AdaGrad problem.
- It is like the AdaGrad problem, but accumulates only the gradients from the most recent iterations (not from the beginning of training).

RMSProp (2/2)

- For each feature $\mathtt{w_i}$, we do the following steps:

$$\mathtt{s_i} = \beta \mathtt{s_i} + (1 - \beta)(\frac{\partial \mathtt{J(w)}}{\partial \mathtt{w_i}})^2$$

$$\mathtt{w_i^{(next)}} = \mathtt{w_i} - \frac{\eta}{\sqrt{\mathtt{s_i} + \epsilon}} \frac{\partial \mathtt{J(w)}}{\partial \mathtt{w_i}}$$

RMSProp (2/2)

- For each feature $\mathtt{w}_\mathtt{i}$, we do the following steps:

$$\mathtt{s}_\mathtt{i} = \beta \mathtt{s}_\mathtt{i} + (1 - \beta)(\frac{\partial \mathtt{J}(\mathtt{w})}{\partial \mathtt{w}_\mathtt{i}})^2$$

$$\mathtt{w}_\mathtt{i}^{(\mathtt{next})} = \mathtt{w}_\mathtt{i} - \frac{\eta}{\sqrt{\mathtt{s}_\mathtt{i} + \epsilon}} \frac{\partial \mathtt{J}(\mathtt{w})}{\partial \mathtt{w}_\mathtt{i}}$$

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer, metrics=["accuracy"])
```

Optimization Algorithms

- Momentum
- Nesterov momentum
- AdaGrad
- RMSProp
- Adam optimization

Adam Optimization (1/3)

- Adam (Adaptive moment estimation) combines the ideas of Momentum optimization and RMSProp.

Adam Optimization (1/3)

- Adam (Adaptive moment estimation) combines the ideas of Momentum optimization and RMSProp.
- Like Momentum optimization, it keeps track of an exponentially decaying average of past gradients.

Adam Optimization (1/3)

- Adam (Adaptive moment estimation) combines the ideas of Momentum optimization and RMSProp.
- Like Momentum optimization, it keeps track of an exponentially decaying average of past gradients.
- Like RMSProp, it keeps track of an exponentially decaying average of past squared gradients.

Adam Optimization (2/3)

$$1.\ \mathbf{m}^{(\mathtt{next})} = \beta_1 \mathbf{m} + (1 - \beta_1)\nabla_{\mathbf{w}} \mathtt{J}(\mathtt{w})$$

$$2.\ \mathbf{s}^{(\mathtt{next})} = \beta_2 \mathbf{s} + (1 - \beta_2)\nabla_{\mathbf{w}} \mathtt{J}(\mathtt{w}) \otimes \nabla_{\mathbf{w}} \mathtt{J}(\mathtt{w})$$

$$3.\ \mathbf{m}^{(\mathtt{next})} = \frac{\mathbf{m}}{1 - \beta_1^{\mathrm{T}}}$$

$$4.\ \mathbf{s}^{(\mathtt{next})} = \frac{\mathbf{s}}{1 - \beta_2^{\mathrm{T}}}$$

$$5.\ \mathbf{w}^{(\mathtt{next})} = \mathbf{w} - \eta \mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon}$$

- $\otimes$ and $\oslash$ represent the element-wise multiplication and division.
- Steps 1, 2, and 5: similar to both Momentum optimization and RMSProp.
- Steps 3 and 4: since $\mathtt{m}$ and $\mathtt{s}$ are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost $\mathtt{m}$ and $\mathtt{s}$ at the beginning of training.

Adam Optimization (3/3)

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer, metrics=["accuracy"])
```

Summary

Summary

- Overfitting
  - Early stopping, $l1$ and $l2$ regularization, max-norm regularization
  - Dropout, data augmentation

- Vanishing gradient
  - Parameter initialization, nonsaturating activation functions
  - Batch normalization, gradient clipping

- Training speed
  - Momentum, nesterov momentum, AdaGrad
  - RMSProp, Adam optimization

Reference

- Ian Goodfellow et al., Deep Learning (Ch. 7, 8)
- Aurélien Géron, Hands-On Machine Learning (Ch. 11)

Questions?