Machine Learning

Lecture 8: Training Deep Neural Networks (Regularization and Optimization)

Instructor: Dr. Farhad Pourkamali Anaraki

Overview

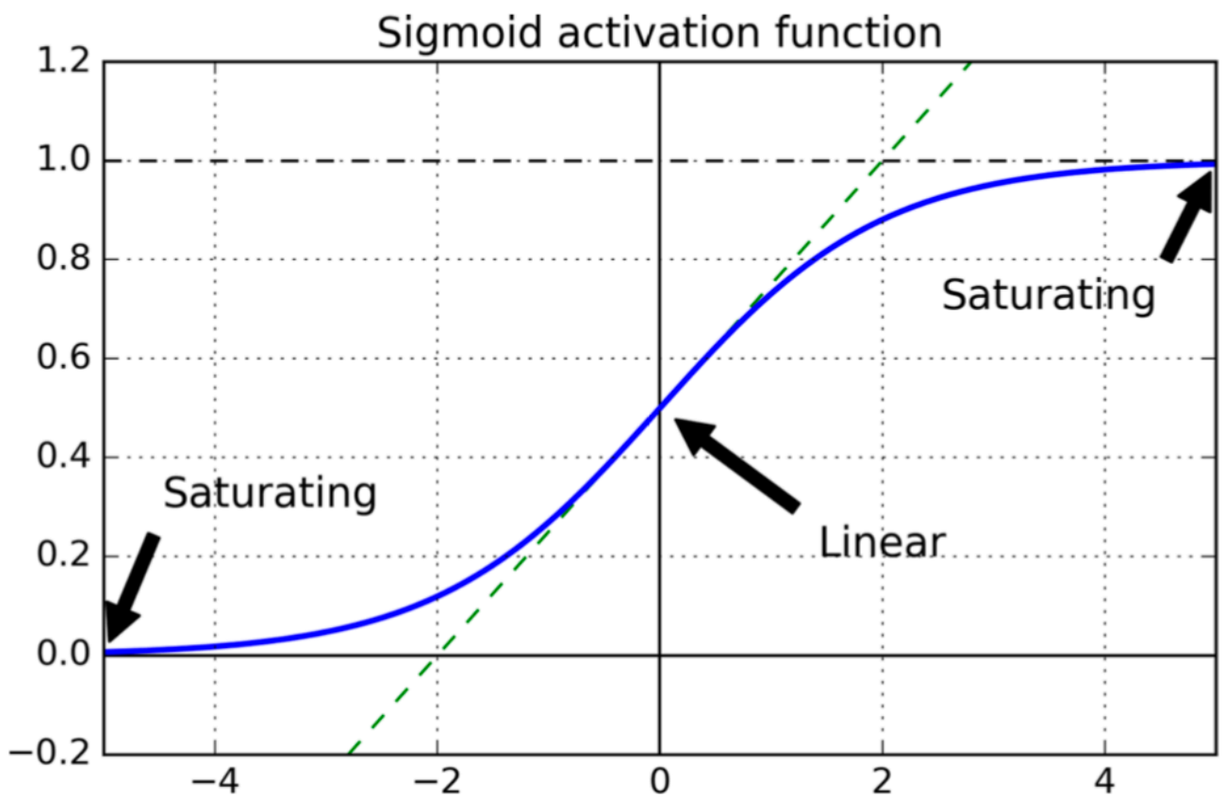
- In this lecture, we review a few techniques to improve the performance of (deep) neural networks
  - Vanishing/exploding gradients
    - Gradients grow smaller and smaller, or larger and larger as we flow backward through the network during training
  - Regularization
    - We might not have enough training data for training a large network or data instances are too noisy
    - Batch normalization and dropout
    - ℓ_1 and ℓ_2 regularization
- Optimizers
  - Various optimization methods can speed up training large neural networks

Vanishing/exploding gradients problem

- When gradients get smaller and smaller, the Gradient Descent update leaves many connection weights unchanged
    - Known as the vanishing gradients problem
- When gradients grow bigger and bigger, some layers get large weight updates and the algorithm diverges
    - Known as the exploding gradients problem
- One of the main reasons deep neural networks were abandoned in 2000s
- It appears there are two main factors
    - Weight initialization
        - Normal distribution with mean 0 and variance 1
    - Sigmoid activation function

Activation function saturation

- When inputs become large (negative or positive), the function saturates at 0 or 1
- Derivative extremely close to 0
- No gradient to propagate back through the network



Glorot and He Initialization

- Goal: the variance of the outputs of each layer to be equal to the variance of its inputs
  - Number of inputs: *fan_in*
  - Number of neurons: *fan_out*
- Let us define: *fan_avg = (fan_in + fan_out)/2*

Equation 11-1. Glorot initialization (when using the logistic activation function)

Normal distribution with mean 0 and variance $\sigma^2 = \frac{1}{fan_{\text{avg}}}$

Or a uniform distribution between $-r$ and $+r$, with $r = \sqrt{\frac{3}{fan_{\text{avg}}}}$

Table 11-1. Initialization parameters for each type of activation function

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, logistic, softmax	$1 / fan_{\text{avg}}$
He	ReLU and variants	$2 / fan_{\text{in}}$
LeCun	SELU	$1 / fan_{\text{in}}$

Keras initializers

- You can choose one of the implemented initializers

```
for name in dir(keras.initializers):
  if not name.startswith("_"):
    print(name, end =", ")
```

```
glorot_normal, glorot_uniform, he_normal, he_uniform, identity, lecun_normal
```

...

- By default, Keras uses Glorot initialization with a uniform distribution

- We can change this to He initialization

  - Method 1

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

  - Method 2

```
init = keras.initializers.VarianceScaling(scale=2., mode='fan_avg',
                                          distribution='uniform')
keras.layers.Dense(10, activation="relu", kernel_initializer=init)
```

Nonsaturating activation functions

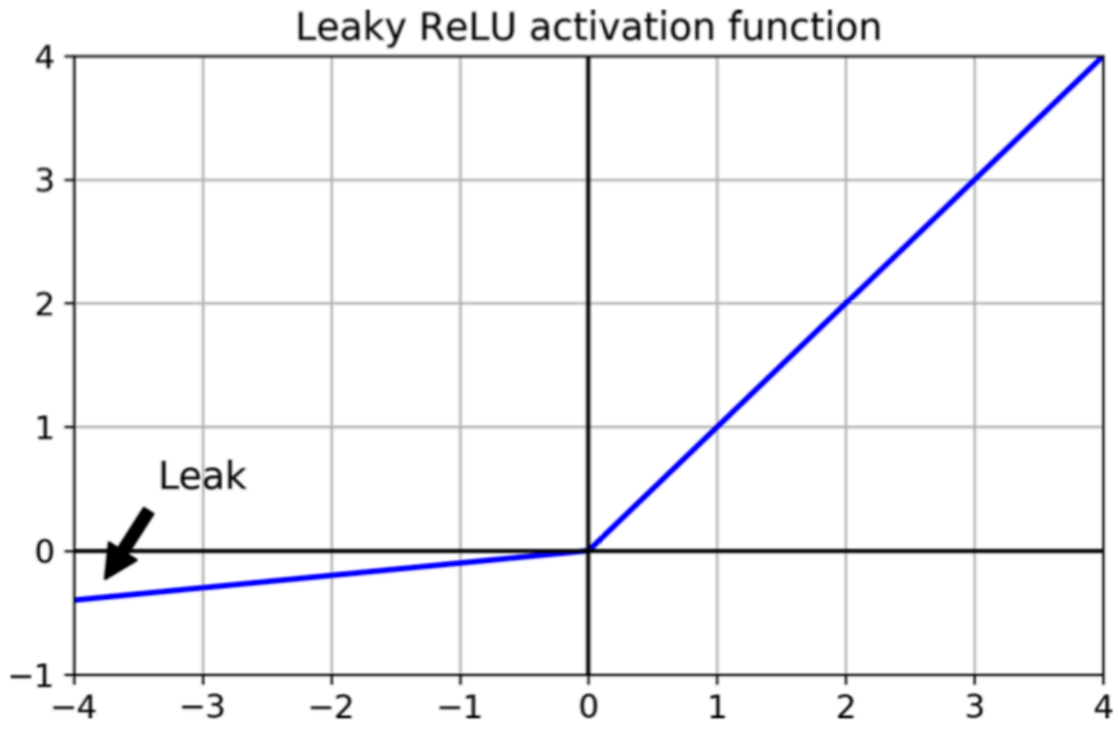
- We know that ReLU activation function does not saturate for positive values and it is fast to compute

- However, ReLU is not perfect!

  - Dying ReLU problem: some neurons effectively “die,” meaning they stop producing anything other than 0

- Solution: Leaky ReLU

$$\text{LeakyReLU}(z) = \max(\alpha z, z), \quad \alpha = 0.01$$

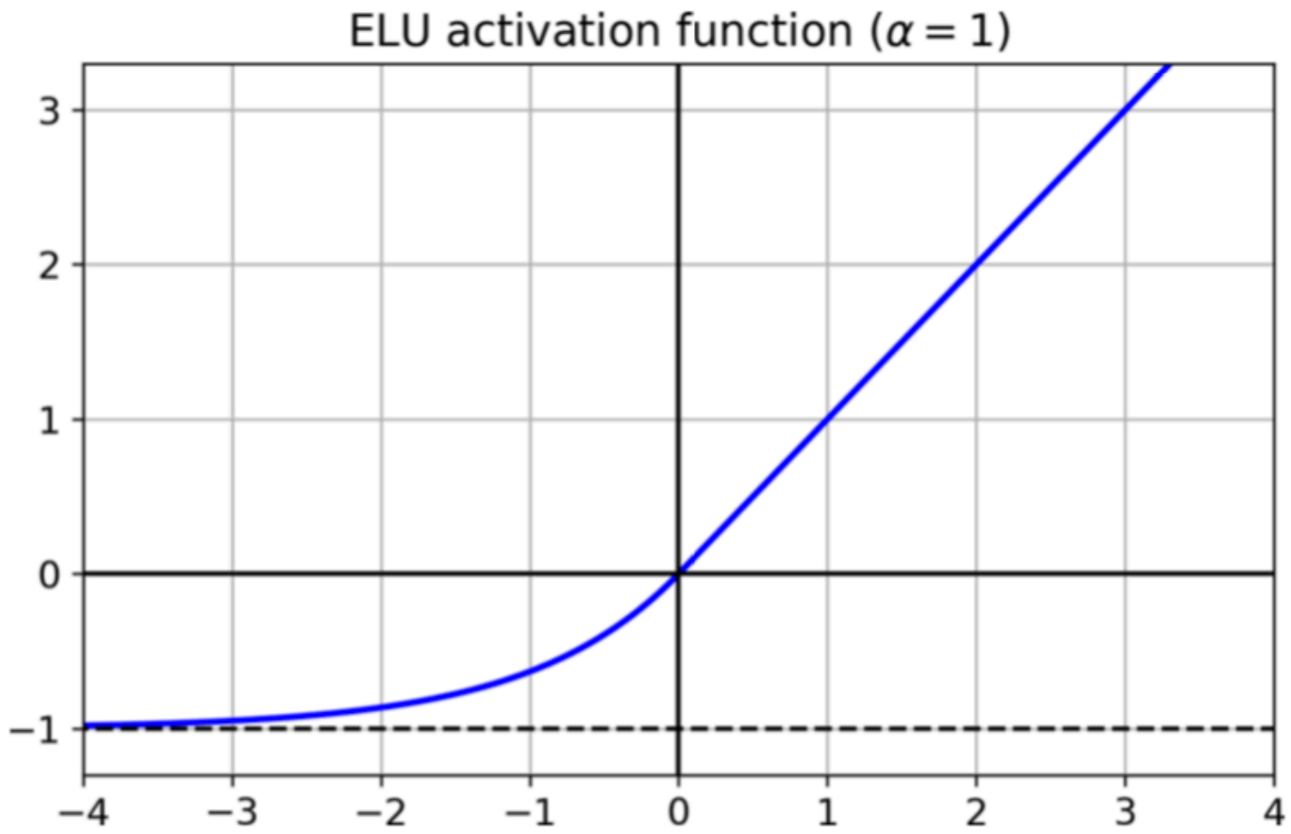


Nonsaturating activation functions

- A new activation function was proposed named exponential linear unit (ELU)

Equation 11-2. ELU activation function

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



- If $\alpha = 1$, the function is smooth everywhere including $z = 0$

Keras activation functions

- We can choose one of the following activation functions

```
for name in dir(keras.activations):
  if not name.startswith("_"):
    print(name, end =", ")
```

```
deserialize, elu, exponential, get, hard_sigmoid, linear, relu, selu,

serialize, sigmoid, softmax, softplus, softsign, swish, tanh
```

- We can also create an additional layer

```
[m for m in dir(keras.layers) if "relu" in m.lower()]
```

```
['LeakyReLU', 'PReLU', 'ReLU', 'ThresholdedReLU']
```

Example

```
(X_train_full, y_train_full), (X_test, y_test) = keras.datasets.fashion_mnist.load_data()
X_train_full = X_train_full / 255.0
X_test = X_test / 255.0
X_valid, X_train = X_train_full[:5000], X_train_full[5000:]
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

```
tf.random.set_seed(42)
np.random.seed(42)

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(),
    keras.layers.Dense(100, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(),
    keras.layers.Dense(10, activation="softmax")
])
```

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=keras.optimizers.SGD(lr=1e-3),
              metrics=["accuracy"])
```

```
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid))
```

Example

```
loss: 1.2819 - accuracy: 0.6229 - val_loss: 0.8886 - val_accuracy: 0.7160

loss: 0.7955 - accuracy: 0.7362 - val_loss: 0.7130 - val_accuracy: 0.7656

loss: 0.6816 - accuracy: 0.7721 - val_loss: 0.6427 - val_accuracy: 0.7898

loss: 0.6217 - accuracy: 0.7944 - val_loss: 0.5900 - val_accuracy: 0.8066

loss: 0.5832 - accuracy: 0.8075 - val_loss: 0.5582 - val_accuracy: 0.8200

loss: 0.5553 - accuracy: 0.8157 - val_loss: 0.5350 - val_accuracy: 0.8236

loss: 0.5338 - accuracy: 0.8224 - val_loss: 0.5157 - val_accuracy: 0.8304

loss: 0.5172 - accuracy: 0.8273 - val_loss: 0.5079 - val_accuracy: 0.8286

loss: 0.5040 - accuracy: 0.8288 - val_loss: 0.4895 - val_accuracy: 0.8390

loss: 0.4924 - accuracy: 0.8321 - val_loss: 0.4816 - val_accuracy: 0.8394
```

Batch normalization

- So far, we discussed initialization techniques and various activation functions

- Batch normalization

  - normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation

  - adds two trainable parameters to each layer, so the normalized output is multiplied by a “standard deviation” parameter (gamma) and add a “mean” parameter (beta)

Equation 11-3. Batch Normalization algorithm

1. $\mathbf{\mu}_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$

2. $\mathbf{\sigma}_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \mathbf{\mu}_B\right)^2$

3. $\widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mathbf{\mu}_B}{\sqrt{\mathbf{\sigma}_B^2 + \varepsilon}}$

4. $\mathbf{z}^{(i)} = \mathbf{\gamma} \otimes \widehat{\mathbf{x}}^{(i)} + \mathbf{\beta}$

Example

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
```

We don't need to standardize the input data when using BN!

2 hidden layers

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
batch_normalization (BatchNo (None, 784)               3136
_________________________________________________________________
dense_7 (Dense)              (None, 300)               235500
_________________________________________________________________
batch_normalization_1 (Batch (None, 300)               1200
_________________________________________________________________
dense_8 (Dense)              (None, 100)               30100
_________________________________________________________________
batch_normalization_2 (Batch (None, 100)               400
_________________________________________________________________
dense_9 (Dense)              (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

=4x784

= (3,136+1,200+400)/2

Example

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=keras.optimizers.SGD(lr=1e-3),
              metrics=["accuracy"])
```

```
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid))
```

```
loss: 0.8293 - accuracy: 0.7221 - val_loss: 0.5539 - val_accuracy: 0.8160
loss: 0.5703 - accuracy: 0.8036 - val_loss: 0.4792 - val_accuracy: 0.8380
loss: 0.5161 - accuracy: 0.8213 - val_loss: 0.4424 - val_accuracy: 0.8490
loss: 0.4789 - accuracy: 0.8314 - val_loss: 0.4212 - val_accuracy: 0.8570
loss: 0.4548 - accuracy: 0.8407 - val_loss: 0.4051 - val_accuracy: 0.8616
loss: 0.4387 - accuracy: 0.8445 - val_loss: 0.3931 - val_accuracy: 0.8632
loss: 0.4255 - accuracy: 0.8502 - val_loss: 0.3829 - val_accuracy: 0.8638
loss: 0.4124 - accuracy: 0.8538 - val_loss: 0.3759 - val_accuracy: 0.8664
loss: 0.4027 - accuracy: 0.8583 - val_loss: 0.3691 - val_accuracy: 0.8676
loss: 0.3925 - accuracy: 0.8613 - val_loss: 0.3630 - val_accuracy: 0.8664
```

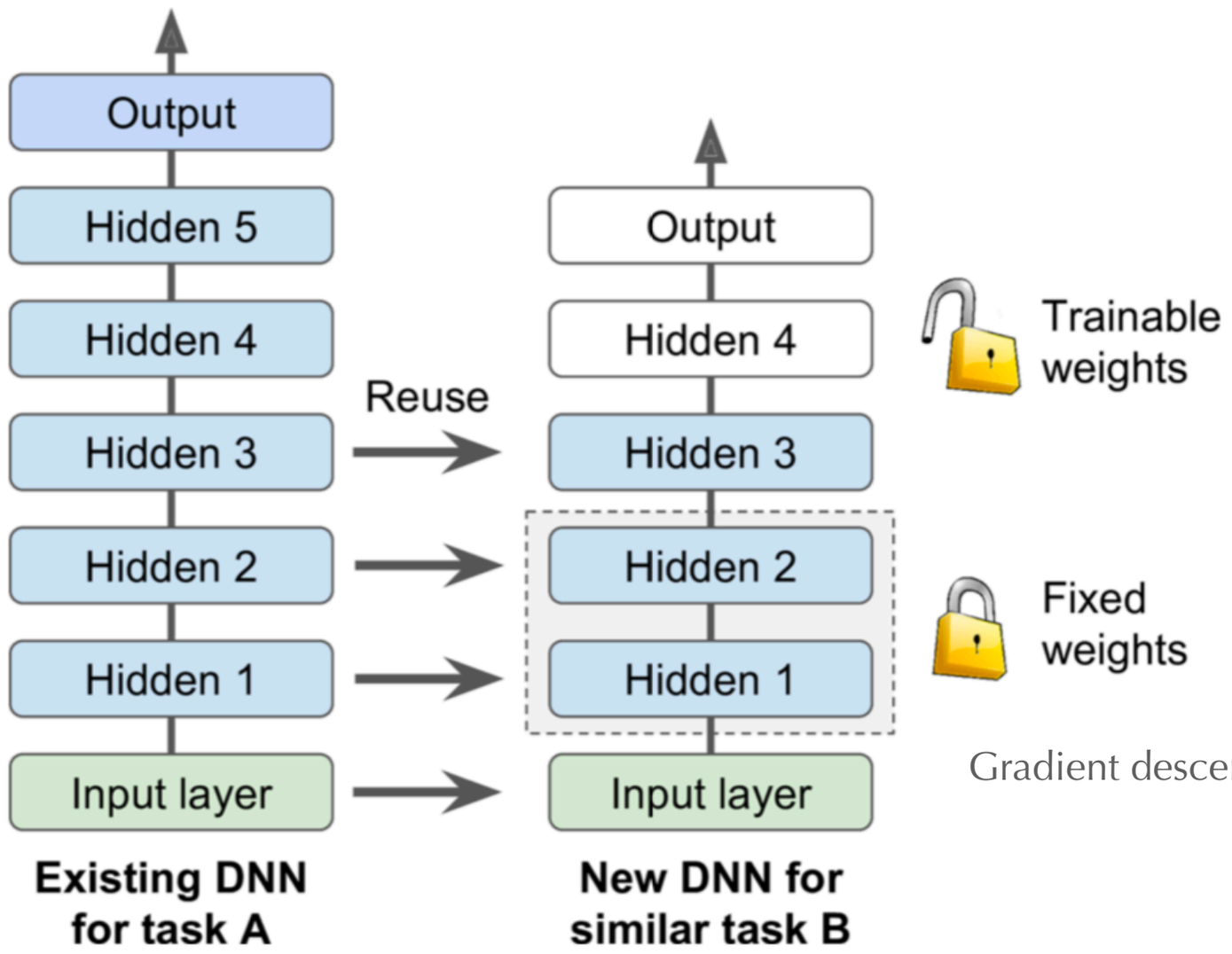
Batch normalization before activation function

- Sometimes applying BN before the activation function works better (there's a debate on this topic)

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("relu"),
    keras.layers.Dense(100, use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Reusing pretrained layers

- In general, it is not a good idea to train large neural networks from scratch
  - We can use an existing neural network and reuse some of the layers
  - Often called “transfer learning”



Gradient descent won’t modify them!

Keras implementation

- Let's look at the Fashion MNIST data set and divide it into two sets
  - `X_train_A`: all images except for sandals and shirts (classes 5 and 6)
  - `X_train_B`: a much smaller training set of just the first 200 images of sandals or shirts
- Objective: we will train a model on set A (classification task with 8 classes), and try to reuse it to tackle set B (binary classification)

```
def split_dataset(X, y):
    y_5_or_6 = (y == 5) | (y == 6) # sandals or shirts
    y_A = y[~y_5_or_6]
    y_A[y_A > 6] -= 2 # class indices 7, 8, 9 should be moved to 5, 6, 7
    y_B = (y[y_5_or_6] == 6).astype(np.float32) # binary classification task: is
    return ((X[~y_5_or_6], y_A),
            (X[y_5_or_6], y_B))

(X_train_A, y_train_A), (X_train_B, y_train_B) = split_dataset(X_train, y_train)
(X_valid_A, y_valid_A), (X_valid_B, y_valid_B) = split_dataset(X_valid, y_valid)
(X_test_A, y_test_A), (X_test_B, y_test_B) = split_dataset(X_test, y_test)
X_train_B = X_train_B[:200]
y_train_B = y_train_B[:200]
```

Keras implementation

- So far, we have two data sets for each model

```
print(X_train_A.shape, X_train_B.shape)
```

```
(43986, 28, 28) (200, 28, 28)
```

- Building model and training

```
model_A = keras.models.Sequential()
model_A.add(keras.layers.Flatten(input_shape=[28, 28]))
for n_hidden in (300, 100, 50, 50, 50):
    model_A.add(keras.layers.Dense(n_hidden, activation="selu"))
model_A.add(keras.layers.Dense(8, activation="softmax"))
```

```
model_A.compile(loss="sparse_categorical_crossentropy",
                optimizer=keras.optimizers.SGD(lr=1e-3),
                metrics=["accuracy"])
```

```
history = model_A.fit(X_train_A, y_train_A, epochs=20,
                      validation_data=(X_valid_A, y_valid_A))
```

Keras implementation

- Let us save the first model

```
model_A.save("my_model_A.h5")
```

- Transfer learning

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile(loss="binary_crossentropy",
                     optimizer=keras.optimizers.SGD(lr=1e-3),
                     metrics=["accuracy"])
```

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                           validation_data=(X_valid_B, y_valid_B))
```

Keras implementation

- Let's train the entire model for a few more epochs

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

model_B_on_A.compile(loss="binary_crossentropy",
                     optimizer=keras.optimizers.SGD(lr=1e-3),
                     metrics=["accuracy"])
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                           validation_data=(X_valid_B, y_valid_B))
```

- Evaluation

```
model_B_on_A.evaluate(X_test_B, y_test_B)
```

```
63/63 [==============================] -
[0.0651376619935035, 0.9934999942779541]
```

Faster optimizers

- Methods that we discussed so far to speed up training
  - Initialization strategies for connection weights
  - Activation functions
  - Batch normalization
  - Reusing parts of a pretrained neural network
- Another important technique is to use a faster optimizer than the regular Gradient Descent optimizer
  - Momentum optimization
  - Nesterov Accelerated Gradient
  - AdaGrad, RMSProp, Adam

Momentum optimization

- Recall the Gradient Descent method updates the weights $\boldsymbol{\theta}$ by directly subtracting the gradient of the cost function $J(\boldsymbol{\theta})$ multiplied by the learning rate η

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla J(\boldsymbol{\theta})$$

- Therefore, gradient descent takes regular steps without taking into account what the earlier gradients were

- Momentum optimization with the hyperparameter β
- Called momentum and must be between 0 and 1 (typical value 0.9)

Equation 11-4. Momentum algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$

- Keras

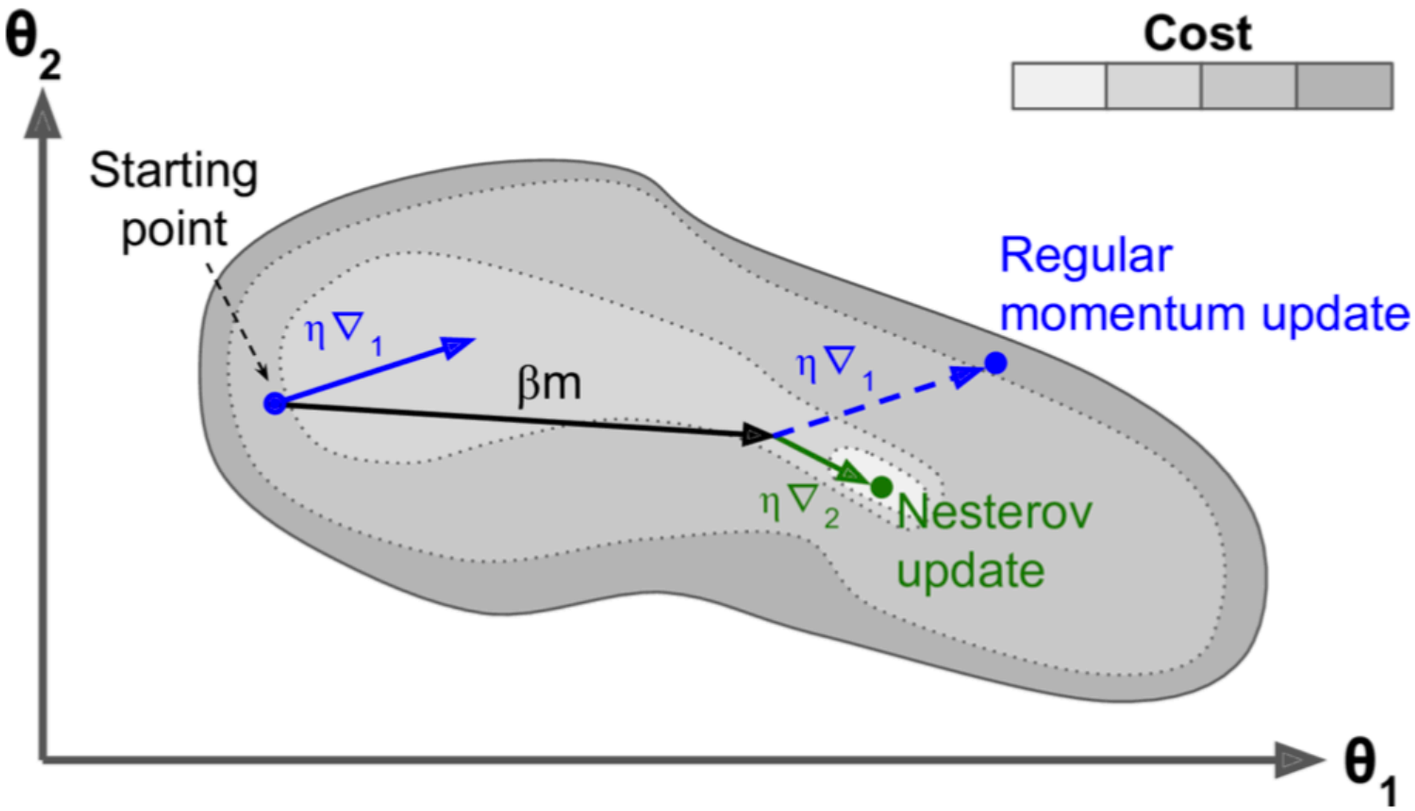
```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

Nesterov Accelerated Gradient

- One small variant to momentum optimization is to measure the gradient of the cost function not at the local position $\boldsymbol{\theta}$ but slightly ahead in the direction of the momentum, i.e., $\boldsymbol{\theta} + \beta\mathbf{m}$

Equation 11-5. Nesterov Accelerated Gradient algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \beta\mathbf{m})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$



- Keras

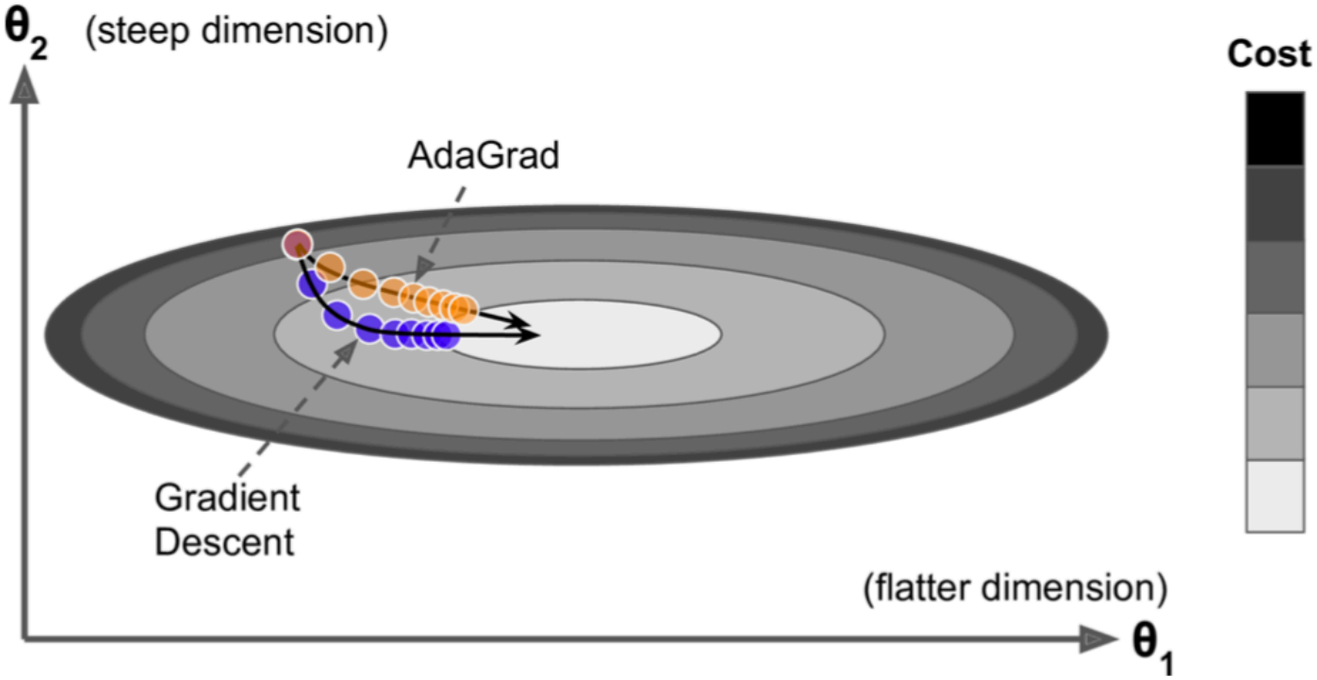
```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

AdaGrad

- Accumulates the square of the gradients for finding an adaptive learning rate, i.e., the algorithm decays the learning rate with regard to parameter θ_i

Equation 11-6. AdaGrad algorithm

1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$



- Sometimes the learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum

RMSProp

- Like AdaGrad but accumulates the gradients from the most recent iterations

Equation 11-7. RMSProp algorithm

1. $\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$

- The decay rate β is typically set to 0.9 (again a new hyperparameter)
- Keras implementation

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

- The *rho* argument corresponds to β in the above equation

Adam

- "Adam" stands for adaptive moment estimation (t represents the iteration number)

Equation 11-8. Adam algorithm

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3. $\widehat{\mathbf{m}} \leftarrow \dfrac{\mathbf{m}}{1 - \beta_1^{\,t}}$
4. $\widehat{\mathbf{s}} \leftarrow \dfrac{\mathbf{s}}{1 - \beta_2^{\,t}}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\, \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \varepsilon}$

- 2 hyperparameters ($\beta_1 = 0.9$ and $\beta_2 = 0.999$)
- Keras implementation

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

  - Requires less time tuning of the learning rate hyperparameter

Regularization

- Just like we explained for simple linear models, you can constrain a neural network's connection weights using ℓ_1 or ℓ_2 norms

- Keras implementation

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01)
```

- We often want to apply the same regularizer to all layers in a network

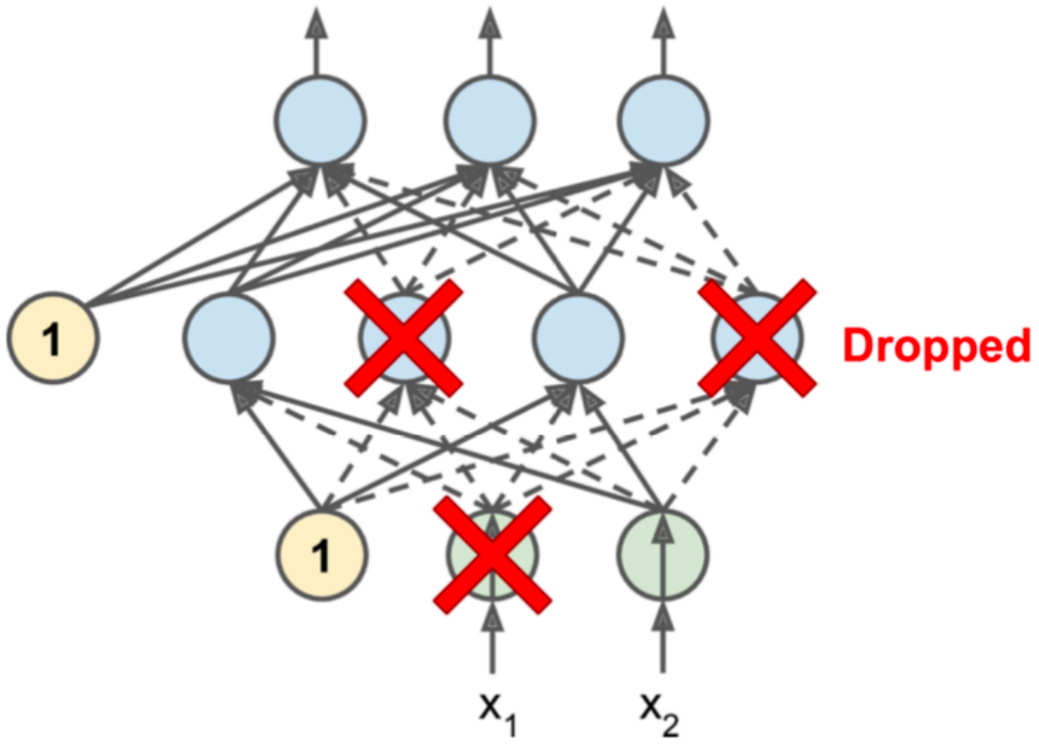
```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

Dropout

- At every training step, every neuron (including the input neurons but excluding the output neurons) has a probability p of being temporarily "dropped out"

  - The hyperparameter p is called the dropout rate and it is typically set between 10% to 50%



- We need to multiply each input connection weight by the *keep probability* $(1-p)$ after training

- If you observe the model is overfitting, you can increase the dropout rate

Keras implementation

- Let's apply dropout regularization before every Dense layer, using a dropout rate of 0.2

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

- For more information, please read chapter 11 of the textbook

Keras documentation

- https://keras.io/api/

Models API

- The Model class
- The Sequential class
- Model training APIs
- Model saving & serialization APIs

Layers API

- The base Layer class
- Layer activations
- Layer weight initializers
- Layer weight regularizers
- Layer weight constraints
- Core layers
- Convolution layers

Optimizers

- SGD
- RMSprop
- Adam
- Adadelta
- Adagrad
- Adamax

Losses

- Probabilistic losses
- Regression losses
- Hinge losses for "maximum-margin" classification