Training Deep Neural Networks

Deep Learning
Stijn Lievens and Sabine Devreese
2023-2024

Possible Problems ....

When training a *deep* neural network, you may encounter the following problems:

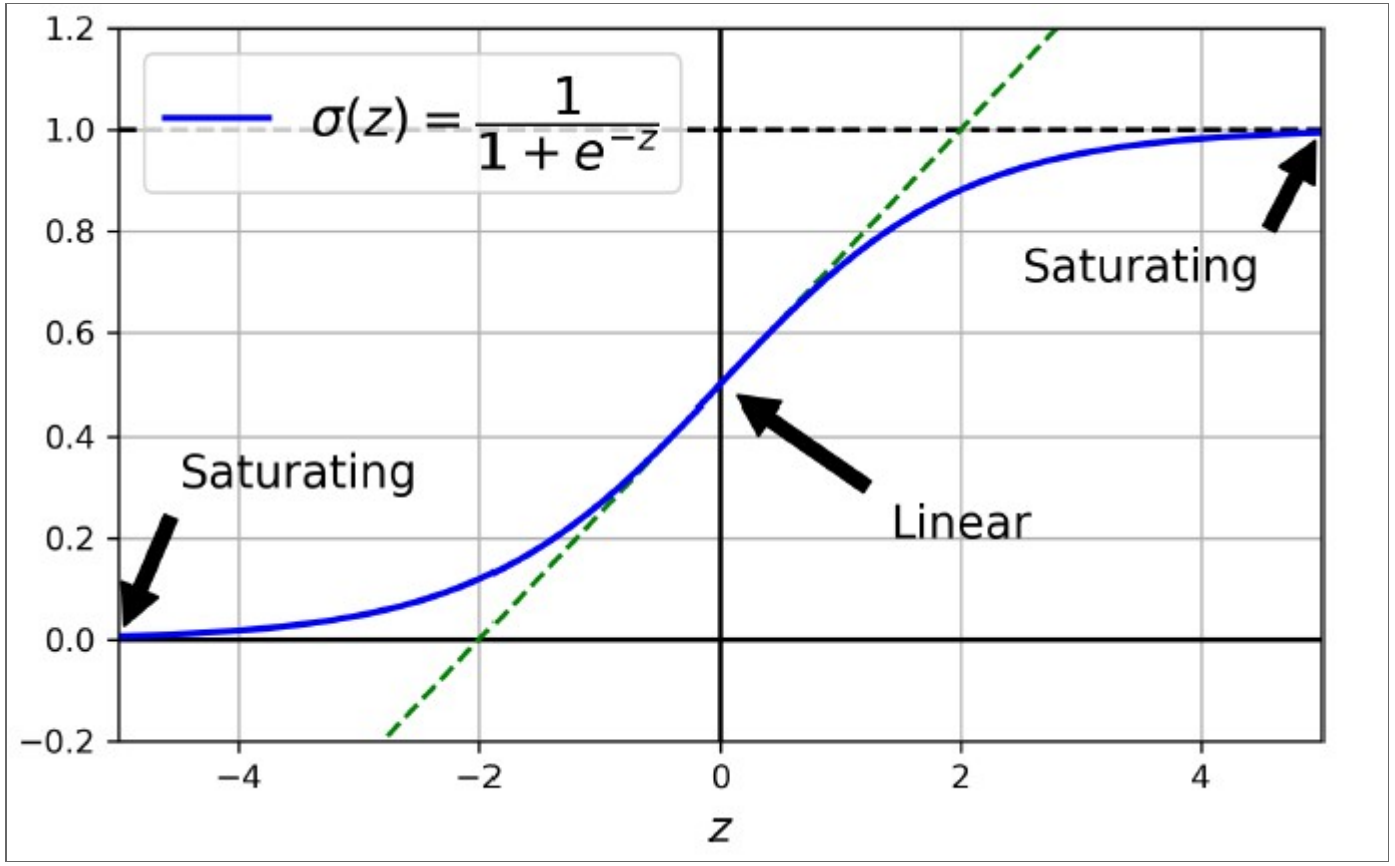
- Gradients growing ever smaller or larger during backpropagation.
- You may not have enough training data for the task.
- Training may be extremely slow.
- You risk overfitting the training data if you use a model with millions of parameters.

11.1 The Vanishing/Exploding Gradient Problems

The Problem

- The backpropagation algorithm works by going from the output layer to the input layer, propagating the error gradient on the way.
- The weights are updated, using the gradient of the cost function with respect to each weight.
- Gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
  - This is called the *vanishing gradient* problem.
  - This leaves the lower layers with very small gradients, so they hardly get updated during training.
- Sometimes the gradients also *explode*. (Typically in RNN)

Sigmoid Activation Function Saturation



Saturation of sigmoid activation function

11.1.1 Glorot and He Initialization

Weight Initialization Strategies

- The initialization strategy used for the connection weights can have a significant impact on the speed at which the algorithm converges.
- The goal is to initialize the weights such that the variance of the outputs of each layer is roughly equal to the variance of its inputs.
  - Think of a chain of amplifiers. Each amplifier should be set up properly in order for your voice to be properly heard at the end of the chain.

Glorot Initialization

- Let $\mathrm{fan}_{\mathrm{avg}} = (\mathrm{fan}_{\mathrm{in}} + \mathrm{fan}_{\mathrm{out}})/2$ be the average number of input and output connections for the layer whose weights are being initialized.
- Then Glorot initialization draws the connection weigths either from
  - a normal distribution with mean 0 and standard deviation $\sigma^2 = 1/\mathrm{fan}_{\mathrm{avg}}$, or
  - a uniform distribution between $-r$ and $+r$, with $r = \sqrt{3/\mathrm{fan}_{\mathrm{avg}}}$.

Variants of Glorot Initialization

Initialization	Activation function	σ^2 (normal)
Glorot	None, tanh, sigmoid, softmax	$\frac{1}{\text{fan}_{\text{avg}}}$
He	ReLU, Leaky ReLU, Swish and variants	$\frac{2}{\text{fan}_{\text{in}}}$
LeCun	SELU	$\frac{1}{\text{fan}_{\text{in}}}$

Note: for uniform distribution between $[-r, r]$, $r = \sqrt{3\sigma^2}$; this actually makes that the variance of the weights is also σ^2.

Using Initialization in Keras

- The default initialization strategy in Keras is Glorot initialization with a uniform distribution.
- You can use the `kernel_initializer` argument to specify a different initialization strategy.

```
import tensorflow as tf

dense = tf.keras.layers.Dense(50, activation="relu",
    kernel_initializer="he_normal") # or "he_uniform"
```

11.1.2 Better Activation Functions

The ReLU Activation Function

- Advantages of ReLU:
  - it does not saturate for positive values
  - it is very fast to compute
- Disadvantage of ReLU:
  - *dying neurons* problem: during training, some neurons effectively die, meaning they stop outputting anything other than 0.
  - if the input for a neuron is negative for all instances in the training set, then this neuron will never fire and the gradient flowing through it will always be zero.

Leaky ReLU

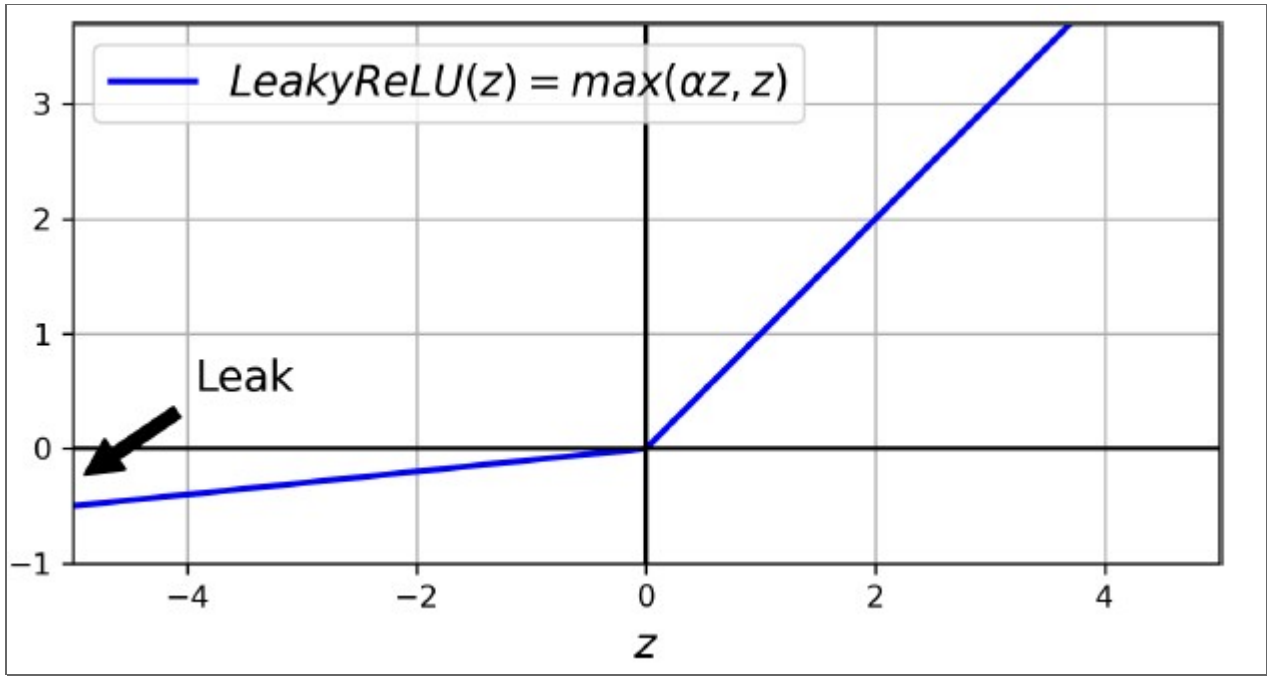
- Remember:

$$\mathrm{ReLU}(z) = \max(0, z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

- Leaky ReLU is defined as:

$$\mathrm{LeakyReLU}_{\alpha}(z) = \max(\alpha z, z) = \begin{cases} \alpha z & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases} \quad \text{with } \alpha > 0.$$

Leaky ReLU



Leaky ReLU

Leaky ReLU

- With Leaky ReLU, the neurons in the hidden layers won’t die.
  - This is because the slope is non-zero everywhere.
  - Neurons may go into a long “coma” but they have a chance to eventually wake up.

Leaky ReLU in Keras

- The LeakyReLU class implements the Leaky ReLU activation function.

```
leaky_relu = tf.keras.layers.LeakyReLU(alpha=0.2)  # defaults to
          alpha=0.3
dense = tf.keras.layers.Dense(50, activation=leaky_relu,
                              kernel_initializer="he_normal")
```

- Note, you should use He initialization when using Leaky ReLU.

Leaky ReLU as a Separate Layer

```
model = tf.keras.models.Sequential([
    # [...]  # more layers
    tf.keras.layers.Dense(50, kernel_initializer="he_normal"),  # no
        activation
    tf.keras.layers.LeakyReLU(alpha=0.2),  # activation as a separate
        layer
    # [...]  # more layers
])
```

The Swish Activation Function

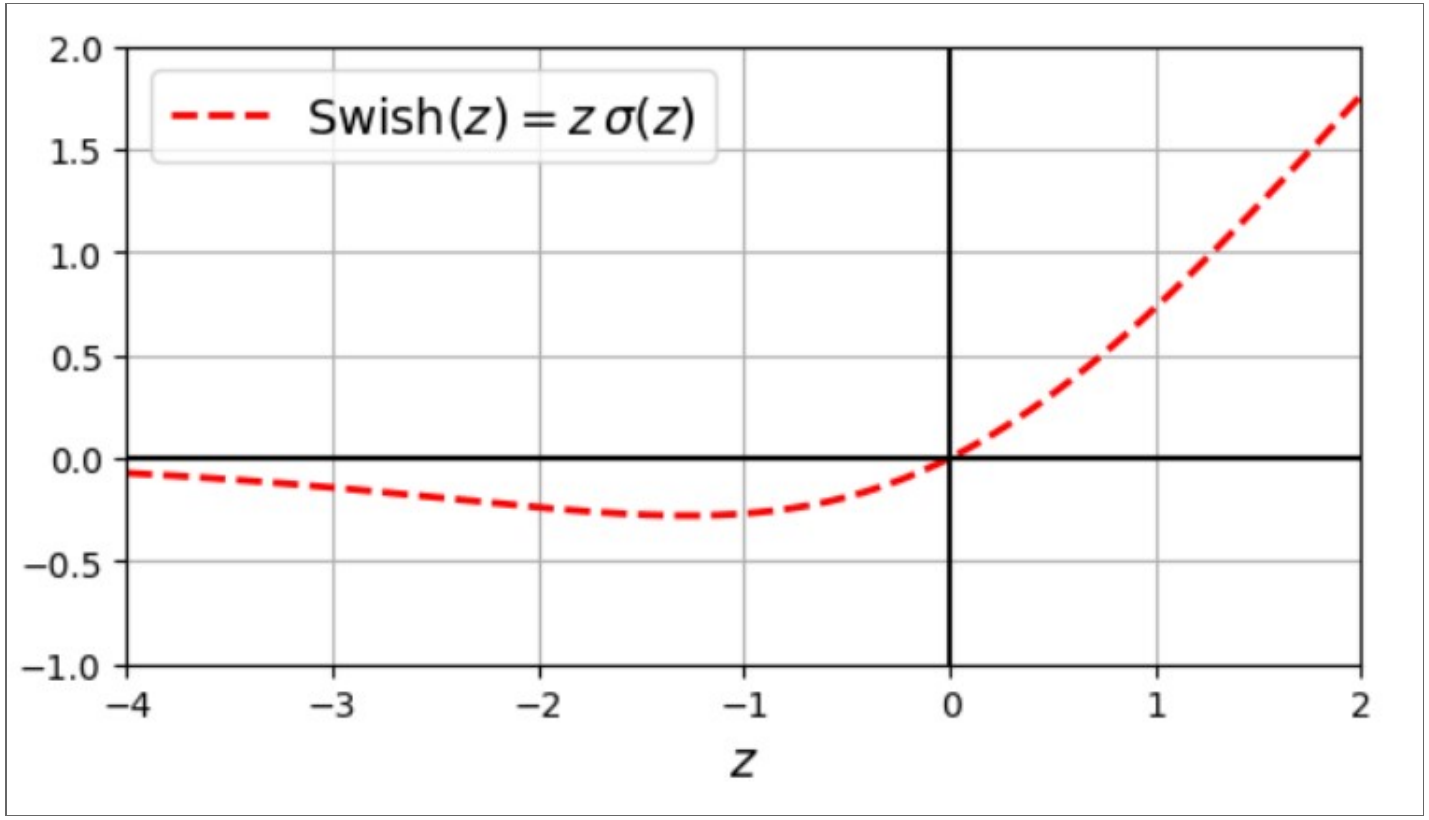
- The Swish activation function is defined as:

$$\text{Swish}(z) = z\,\text{sigmoid}(z).$$

- The function can be generalized to

$$\text{Swish}_\beta(z) = z\,\text{sigmoid}(\beta z).$$

The Swish Activation Function



Swish activation function

Swish in Keras

- Keras supports Swish out of the box: simply use `activation="swish"`.

Advice on Using Activation Functions

- ReLU is a good default for simple tasks.
- For more complex tasks, Swish is probably a better default.
- If you care a lot about runtime latency, then you may prefer leaky ReLU.
  (See page 366 in the book for more details)

11.1.3 Batch Normalization

Problem Statement

- Using He initialization and ReLU (or a variant) can significantly reduce the vanishing/exploding gradients problems at the beginning of training, but it doesn't guarantee that they won't come back during training.
- **Batch normalization** (BN) is a technique that addresses this issue.

Central Idea

- Add an operation in the model just before or after the activation function of each hidden layer.
  - This operation will zero-center and normalize each input, then scale and shift the result using two new parameter vectors per layer: one for scaling, the other for shifting.
  - In other words, this operation lets *the model learn the optimal scale and mean of each of the layer's inputs*

Formulas for BN

- Suppose $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m_B)} \in \mathbb{R}^n$ are the inputs of a given layer for a mini-batch of size m_B.
- Compute the mean $\mu_B \in \mathbb{R}^n$ and variance $\sigma_B^2 \in \mathbb{R}^n$ of each input dimension over the mini-batch:

$$\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)} \quad \text{and} \quad \sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \mu_B \right)^2.$$

Formulas for BN

- Normalize the inputs:

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

with ϵ a tiny positive number (e.g., 10^{-5}) to avoid division by zero. The *smoothing term*

Formulas for BN

- Finally, scale and shift the result:

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

- γ and β are both vectors in $\mathbb{R}^n$. They are parameter vectors *learned during training*.

BN at Test/Inference Time

- At test time, there is no mini-batch to compute the empirical mean and variance of each input dimension.
  - I.e. we can't compute μ_B and σ_B^2.
- Solution: during training estimates for (the final) μ and σ^2 are computed.
- An *exponential moving average* of these estimates is kept track of during training.
  - These are the *non learnable parameters* of the BN layer.

Benefits of BN

- The vanishing gradient problem is strongly reduced.
- Networks are less sensitive to the weight initialization.
- Much larger learning rates can be used, speeding up the learning process.
- BN acts like a regularizer, reducing the need for other regularization techniques.

Disadvantages of BN

- BN adds some complexity to the model.
  - The model also behaves differently during training and at test time.
- If you use BN, it will make slower predictions due to the additional computations required at each layer.

Fusing BN with the Previous Layer

- It is often possible to fuse the BN layer with the previous layer for inference
  - E.g. if the previous layer computes $\mathbf{XW} + \mathbf{b}$.
  - The BN layer will then compute

$$\gamma \otimes ((\mathbf{XW} + \mathbf{b} - \mu)/\sigma) + \beta = \mathbf{X}(\gamma \otimes \mathbf{W}/\sigma) + \gamma \otimes (\mathbf{b} - \mu)/\sigma + \beta$$

- So, the BN layer can be fused with the previous layer by updating the previous layer's weights and biases.

$$\mathbf{W}' = (\gamma \otimes \mathbf{W}/\sigma) \quad \text{and} \quad \mathbf{b}' = \gamma \otimes (\mathbf{b} - \mu)/\sigma + \beta$$

BN in Keras

- Keras supplies a `BatchNormalization` layer that does all this for you.
- Using BN layer after the activation function:

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

BN in Keras

- The `summary` of the model will now also show *non-trainable parameters*:

```
model.summary()
```

gives

```
Layer (type)                 Output Shape              Param #
...
batch_normalization (BatchNo (None, 784)               3136
....
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

BN in Keras

- The `BatchNormalization` layer adds four parameters per input: γ, β, μ, and σ.
  - For the first BN layer, there are 784 inputs, so $4 \times 784 = 3136$ parameters.
  - Half of these are consider trainable, namely γ and β. The other half are not.
- The other BN layers have 300 and 100 inputs, so they add $4 \times 300 + 4 \times 100 = 1600$ parameters each.
- Total number of non-trainable parameters is thus $(3136 + 1200 + 400)/2 = 2368$.

BN in Keras

- If you want to use BN before the activation function, you must add the activation function manually:

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

- Note that in this case we can use `use_bias=False` in the `Dense` layers. Why?

Hyperparameters of BN

- One hyperparameter is `momentum`, which is used to compute the moving averages.

$$\hat{\mathbf{v}} \leftarrow \text{momentum} \times \hat{\mathbf{v}} + (1 - \text{momentum}) \times \mathbf{v}$$

  - A good momentum value is typically close to 1, e.g. 0.9, 0.99, or 0.999.

Hyperparameters of BN

- The `axis` hyperparameter determines which axis should be normalized.
  - If defaults to -1, which means the last axis.
  - When the batch has shape `(batch size, features)` then each feature is normalized independently.
  - When the batch has shape `(batch size, height, width)` then by default the BN layer will compute `width` means and variances.
    - If you want to normalize each pixel independently, you must set `axis=[1, 2]`.

11.1.4 Gradient Clipping

Gradient Clipping

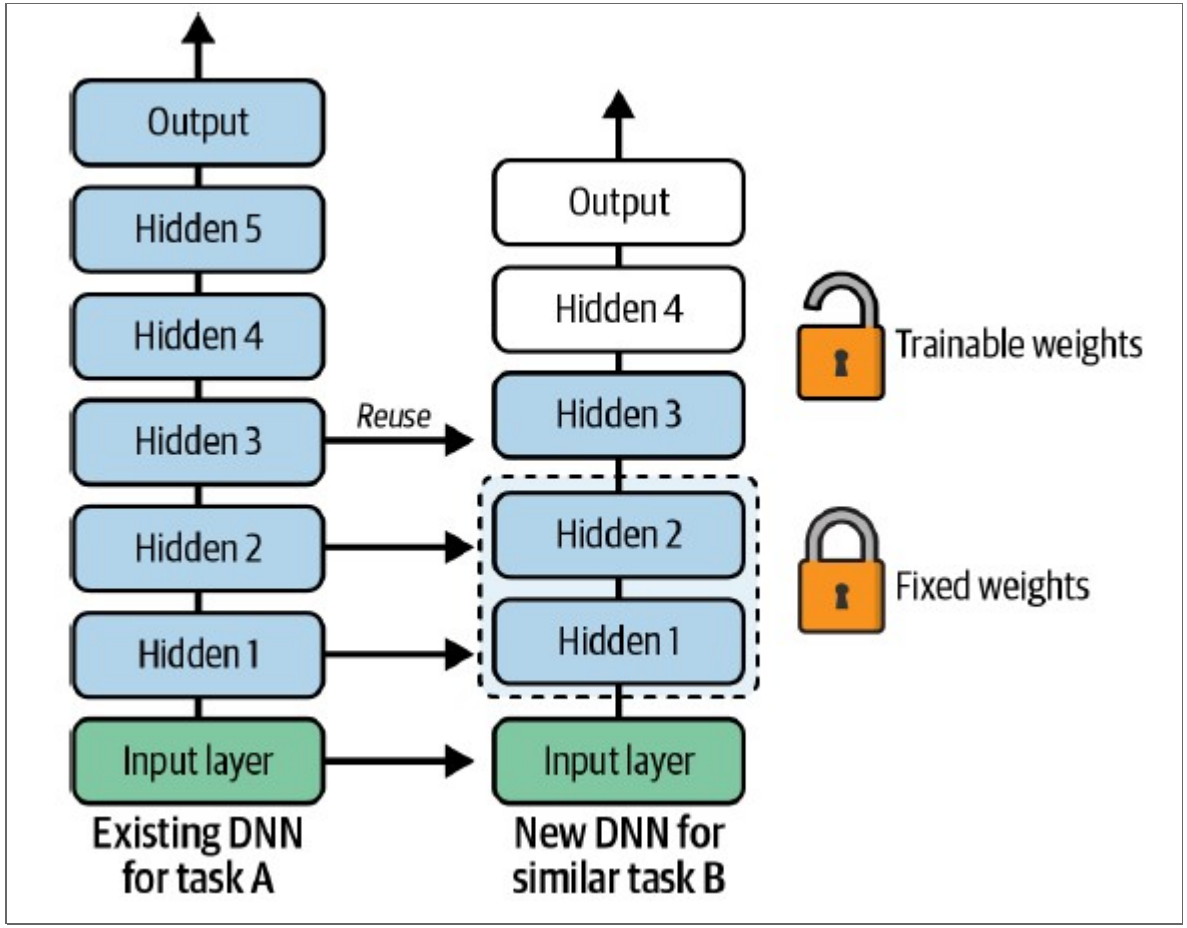
- Gradient clipping is a technique to prevent exploding gradients in RNNs.
- Passing `clipvalue=1.0` to the optimizer will clip the components of the gradient to the range `[-1.0, 1.0]`.
  - This may change the direction of the gradient.
- Instead passing `clipnorm=1.0` will clip the norm of the gradient to 1.0.
  - This will preserve the direction of the gradient.

11.2 Reusing Pretrained Layers

General Idea

- It is generally not a good idea to train a very large DNN from scratch.
- Instead, try to find an existing neural network that accomplishes a similar task to the one you are trying to tackle.
  - Then reuse the lower layers of this network.
  - This is called **transfer learning**.

Transfer Learning: Picture



Reusing pretrained layers

Transfer Learning

- Replace the output layer of the pretrained network with a new one.
  - It is probably not useful for the new task, it may not even have the correct number of neurons.
- Maybe some upper hidden layers should be replaced as well.
- Step 1: freeze all the layers of the pretrained model.
  - This means that their weights will not be updated during training.
- Step 2: Unfreeze some of the reused layers.
  - This allows them to be modified during training.
  - Typically, you will use a small learning rate to avoid wrecking the reused weights.

11.2.1 Transfer Learning with Keras

Transfer Learning with Keras

- Suppose we have a model trained on a similar task.

```
# Existing model
model_A = tf.keras.models.load_model("my_model_A")
# Remove output layer
model_B_on_A = tf.keras.Sequential(model_A.layers[:-1])
 # Add new output layer
model_B_on_A.add(tf.keras.layers.Dense(1, activation="sigmoid"))
```

- Note: `model_A` and `model_B_on_A` now share some layers. If you train `model_B_on_A`, then `model_A` will also be affected.

Transfer Learning with Keras

- Alternatively, you can clone `model_A` before you reuse its layers.

```
model_A_clone = tf.keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
model_B_on_A = tf.keras.Sequential(model_A_clone.layers[:-1])
model_B_on_A.add(tf.keras.layers.Dense(1, activation="sigmoid"))
```

- `clone_model` only clones the architecture, not the weights. Use `set_weights` to copy the weights.

Freezing the Reused Layers

- To avoid wrecking the reused weights during training, you must freeze them.

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

- Note: you must compile the model after you freeze or unfreeze layers!

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                     metrics=["accuracy"])
```

Train for a Few Epochs

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

- Optionally, unfreeze the reused layers, and continue training for more epochs:

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

optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                     metrics=["accuracy"])
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                           validation_data=(X_valid_B, y_valid_B))
```

Warning on Transfer Learning

This example was cherry picked to yield good performance. Transfer learning works best with deep convolutional neural networks.

11.2.2 Unsupervised Pretraining

11.2.3 Pretraining on an Auxiliary Task

11.3 Faster Optimizers

Solutions to Slow Training

Training a very large deep neural network can be painfully slow.
Here are some options to speed up training:

- apply a good initialization strategy for the connection weights,
- use a good activation function (e.g., ReLU),
- apply batch normalization,
- reuse parts of a pretrained network.
  Using a *better optimizer* can also speed up training considerably.

Regular Gradient Descent

- Gradient descent performs the following update to optimize a cost function $J(\theta)$:

$$\theta \rightarrow \theta - \eta \nabla_\theta J(\theta),$$

  where η is the learning rate.
- Thus:
  - gradient descent only cares about the current gradient.
  - it takes small steps when the gradient is small, and big steps when the gradient is big, but it never picks up speed.

Optimizers Overview

- There are many different optimization algorithms available.
  - We won't go into the mathematical details of each one.
- Each optimizer is available in TensorFlow via the `tf.keras.optimizers` package.

Optimizers Overview

Class	Convergence speed	Convergence quality
`SGD`	*	***
`SGD(momentum=...)`	**	***
`SGD(momentum=..., nesterov=True)`	**	***
`Adagrad`	***	* (stops too early)
`RMSprop`	***	** or ***
`Adam`	***	** or ***
`AdaMax`	***	** or ***
`Nadam`	***	** or ***
`AdamW`	***	** or ***

Optimizers, * = bad, ** = average, *** = good

Example Optimizer in Keras

```
# Nesterov Accelerated Gradient
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9,
                                    nesterov=True)

# RMSProp
optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9)

# Adam
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9,
                                     beta_2=0.999)
```

11.4 Learning Rate Scheduling

11.5 Avoiding Overfitting Through Regularization

Regularization

- Deep neural networks have so many parameters that they are prone to overfitting the training set.
- **Regularization** is any technique that tries to combat this overfitting.
- Early stopping is a form of regularization.
- Other forms of regularization are:
  - ℓ_1 and ℓ_2 regularization,
  - dropout,
  - max-norm regularization,
  - data augmentation.

11.5.1 ℓ_1 and ℓ_2 Regularization

ℓ_1 and ℓ_2 Regularization

- This is like regularization in linear models where you add a regularization term to the cost function.

$$J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} \theta_i^2$$

ℓ_1 and ℓ_2 Regularization

- In Keras you achieve this by specifying the `kernel_regularizer` argument when creating a layer:

```
layer = tf.keras.layers.Dense(100, activation="relu",
                              kernel_initializer="he_normal",

          kernel_regularizer=tf.keras.regularizers.l2(0.01))
```

- This will add a regularization term to the cost function equal to the sum of the squared values of the weights, multiplied by a constant (0.01 in this case).
- You can also use `tf.keras.regularizers.l1` and `tf.keras.regularizers.l1_l2`.

Using `partial` to Avoid Repetition

- Very often you will want to apply the same regularizer to many layers in your network.
  - This can lead to a lot of repetition.
- One solution is to use Python's `functools.partial()` function:

```
from functools import partial

RegularizedDense = partial(tf.keras.layers.Dense,
                           activation="relu",
                           kernel_initializer="he_normal",

          kernel_regularizer=tf.keras.regularizers.l2(0.01))

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(100),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax")
])
```

Final Remark

Do not use ℓ_2 regularization with the Adam optimizer and its variants. Use the AdamW optimizer instead.

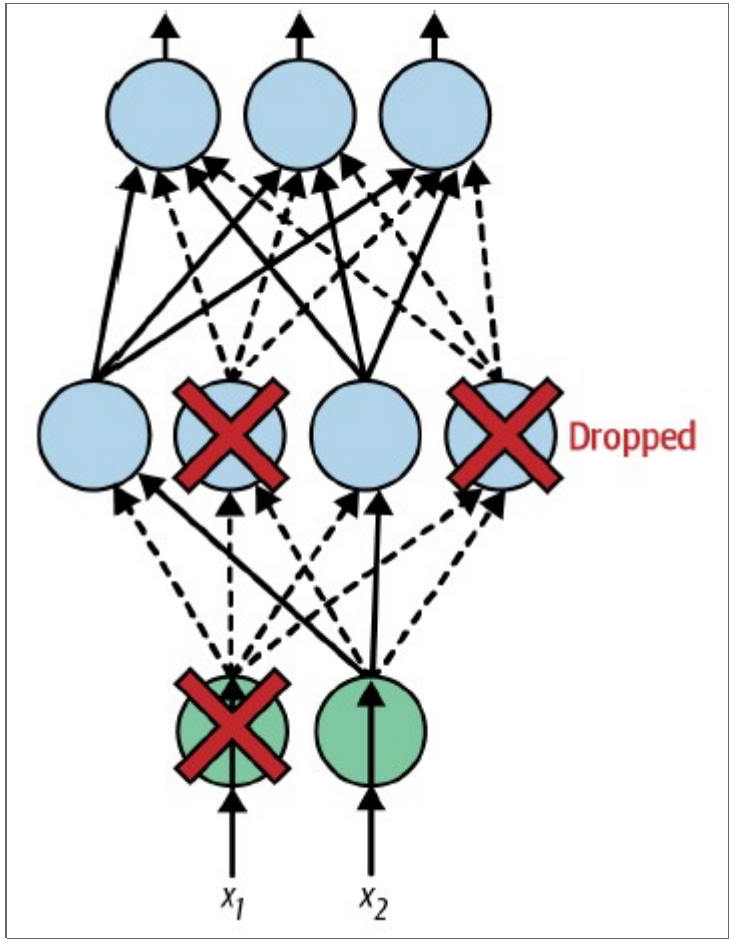
11.5.2 Dropout

Dropout

Dropout is one of the most popular regularization techniques for deep neural networks.

- it is a simple algorithm: at every training step, every neuron (including the input neurons, but always excluding the output neurons) has a probability p of being temporarily “dropped out”
- it will the be entirely ignored during this training step, but it may be active during the next step
- the hyperparameter p is called the dropout rate, and it is typically set between 10% and 50%

Dropout



Dropped out neurons

Dropout as a Regularizer

- We can understand the power of dropout by realizing that a unique neural network is generated at each training step.
- Also, neurons cannot rely on the presence of particular other neurons (because they may be dropped out at any time).

Dropout in Keras

- To implement dropout using Keras, you can use the `keras.layers.Dropout` layer.
- During training, it randomly drops some inputs (setting them to 0).
- During inference, this layer doesn't do anything at all.
- If you observe that the model is overfitting, you can increase the dropout rate (and vice versa).

Dropout in Keras

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

Training and Validation Loss

Comparing the training loss (during training and with dropout) to the validation loss (without dropout) can be misleading! To check for overfitting compare the training and the validation loss both without dropout!

11.5.3 Monte Carlo (MC) Dropout

Monte Carlo (MC) Dropout

- Dropout is only active during training.
- However, by running the model many times with dropout active
  - this acts as an ensemble of models
  - we can get a sense of the uncertainty of the model's predictions.

Monte Carlo (MC) Dropout in Keras

```
y_probas = np.stack([model(X_test, training=True)
                     for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

- Suppose `X_test` is a batch of 10000 test instances and that we have 10 classes.
- A single call to the model's `__call__()` method will return a tensor of shape `(10000, 10)`.
- We do this 100 times, hence `y_probas` will have shape `(100, 10000, 10)`.

Monte Carlo (MC) Dropout in Keras

- We then compute the mean of the 100 predictions for each of the 10000 instances.
- So, finally `y_proba` will have shape `(10000, 10)`.

> *Averaging over multiple predictions with dropout turned on will in general give a more reliable result.*

Advanced Implementation Note

- We have forced `training=True` in the call to the model's `__call__()` method.
  - However, this will not work if the model contains other layers that behave differently during training and during inference (e.g., `BatchNormalization`).
- Instead, you could implement an `MCDropout` layer.

Advanced Implementation Note

```
class MCDropout(tf.keras.layers.Dropout):
    def call(self, inputs, training=None):
        return super().call(inputs, training=True)
```

Change the `Dropout` layers in the model to `MCDropout` layers.

```
Dropout = tf.keras.layers.Dropout
mc_model = tf.keras.Sequential([
    MCDropout(layer.rate) if isinstance(layer, Dropout) else layer
    for layer in model.layers
])
mc_model.set_weights(model.get_weights())
```

11.6 Summary and Practical Guidelines

Default DNN Configuration

Hyperparameter	Default Value
Kernel initializer	He initialization
Activation function	ReLU if shallow; Swish if deep
Normalization	None if shallow; Batch Normalization if deep
Regularization	Early stopping; weight decay if needed
Optimizer	Nesterov accelerated gradients or AdamW
Learning rate schedule	Performance scheduling or 1cycle

Configuration of Self-Normalizing Networks

Hyperparameter	Default Value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Nesterov accelerated gradients
Learning rate schedule	Performance scheduling or 1cycle

Note: the input features must be normalized!

Exceptions to the Above

- For sparse models, use ℓ_1 regularization.
- For low-latency models, use fewer layers, use a fast activation function like `ReLU` activation function, fold batch-norm layers into previous layers after training.
- For risk-sensitive application (or when inference latency is not very important), use Monte Carlo (MC) dropout to boost performance and get more reliable estimates of uncertainty.