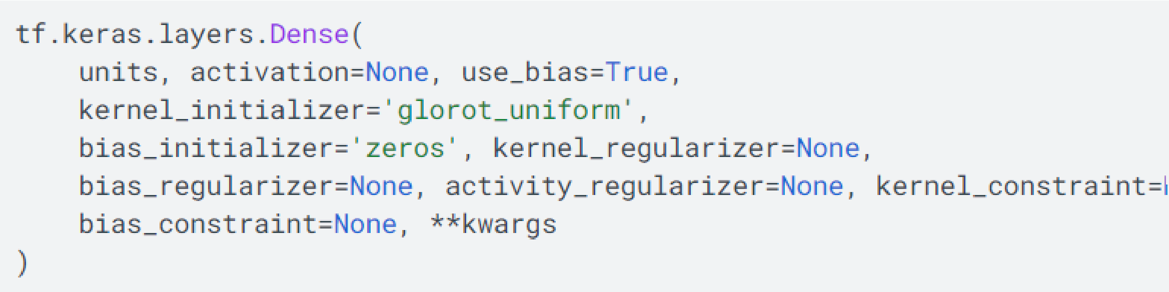
Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd edition by Aurélien Geron Chapter 11

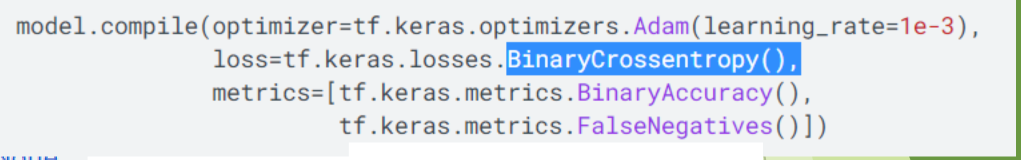
San Diego Machine Learning
2022 JAN 15
Discussion Leader: Robert Kraig

CHAPTER 11 OUTLINE

- Vanishing / Exploding Gradients
- Reusing Pretrained Layers
- Faster Optimizers
- Regularization to Avoid Overfitting

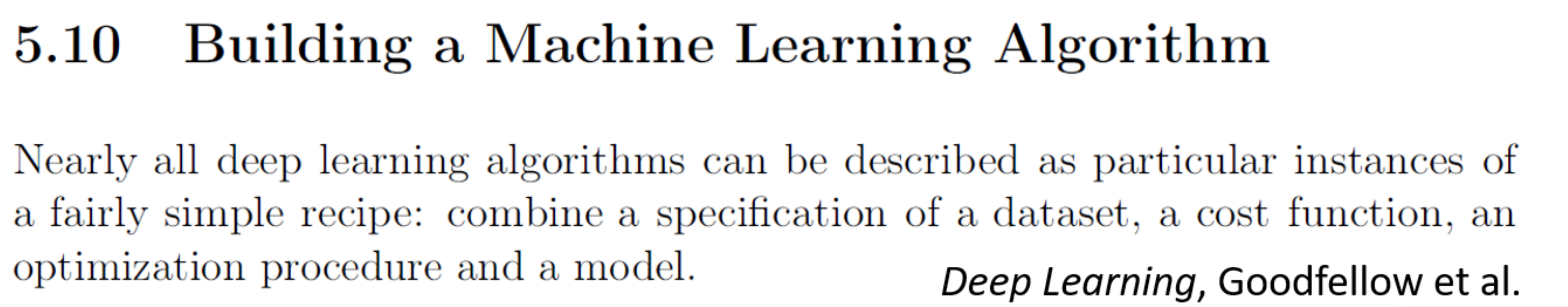


```
tf.keras.layers.Dense(
    units, activation=None, use_bias=True,
    kernel_initializer='glorot_uniform',
    bias_initializer='zeros', kernel_regularizer=None,
    bias_regularizer=None, activity_regularizer=None, kernel_constraint=None,
    bias_constraint=None, **kwargs
)
```



```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=[tf.keras.metrics.BinaryAccuracy(),
                       tf.keras.metrics.FalseNegatives()])
```

TensorFlow
Hub



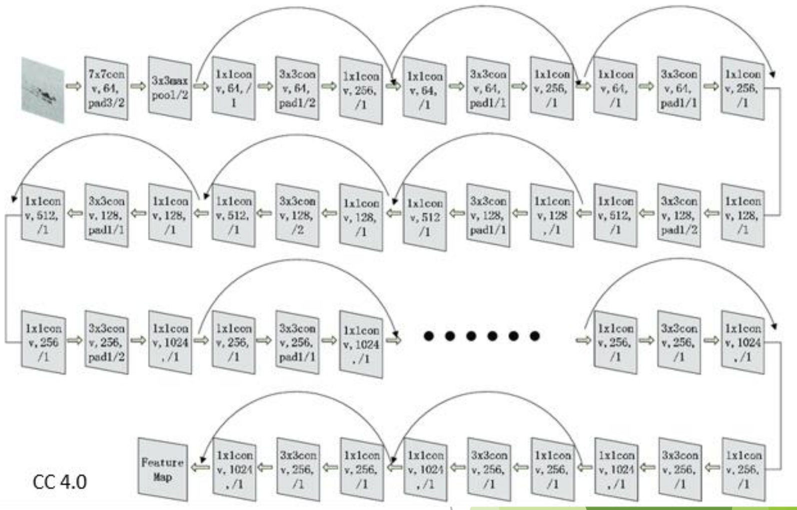
5.10 Building a Machine Learning Algorithm

Nearly all deep learning algorithms can be described as particular instances of a fairly simple recipe: combine a specification of a dataset, a cost function, an optimization procedure and a model.

Deep Learning, Goodfellow et al.

1. Vanishing and Exploding Gradients

- Glorot and Bengio (2010):
    - Vanishing Gradients Expected in early NNs
        - Standard Normal Initialization
        - Sigmoid Activation: Small Derivatives



CC 4.0

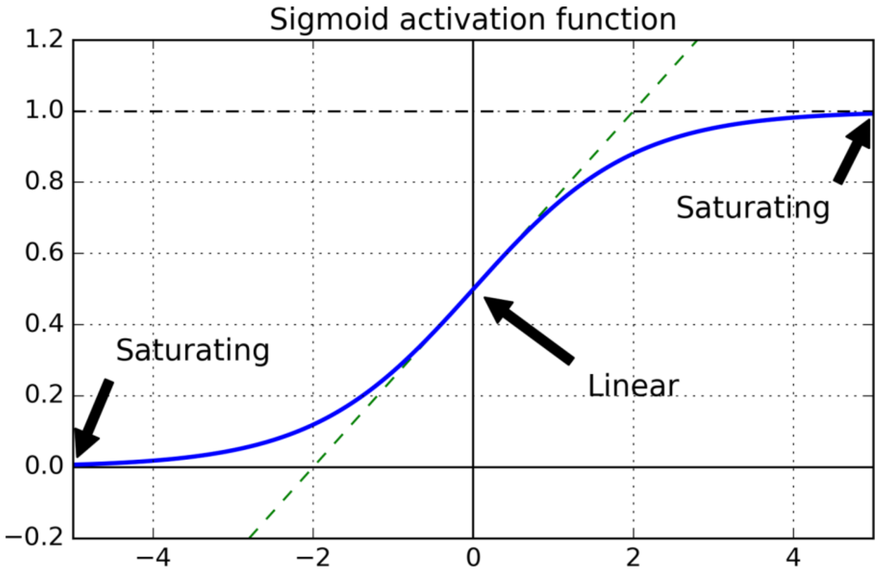
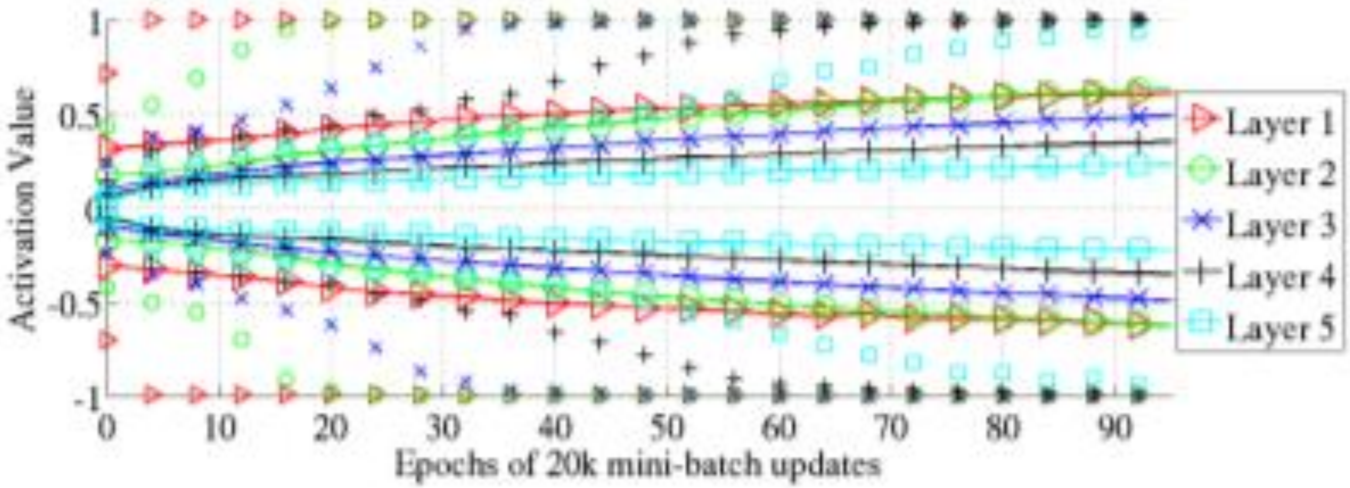


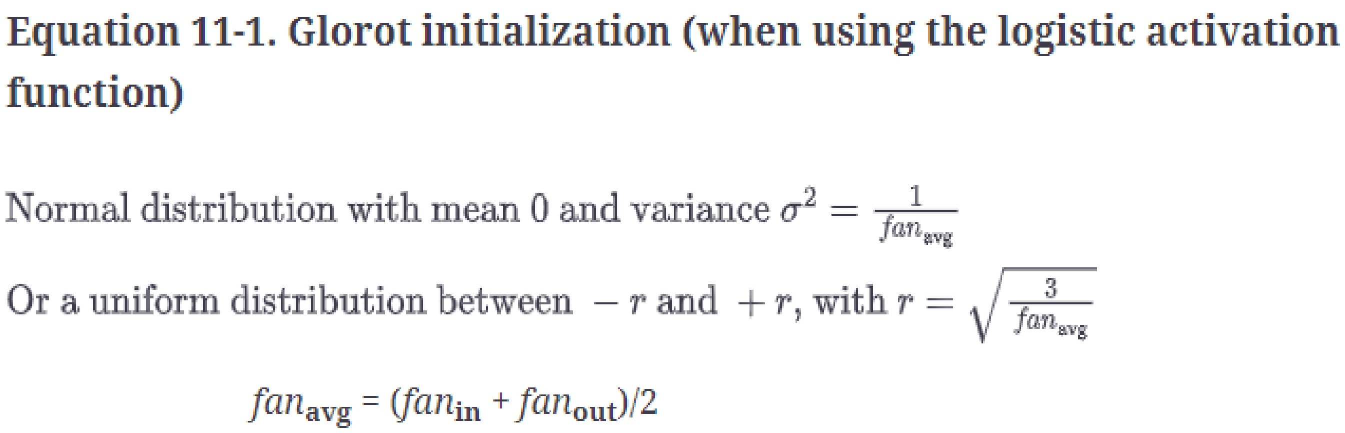
Figure 11-1. Logistic activation function saturation



Glorot and Bengio

Initializers

- Glorot and Bengio observed that, for stability:
  - Each layer should have equal variance of input and output activations (Forward)
  - Each layer should have equal variance of input and output gradients (Back-Prop)



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 $r = \sqrt{\frac{3}{fan_{\text{avg}}}}$

$$fan_{\text{avg}} = (fan_{\text{in}} + fan_{\text{out}})/2$$

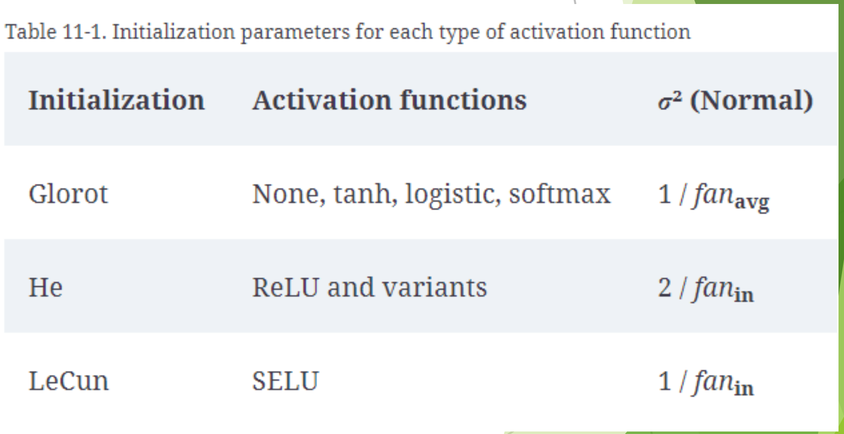
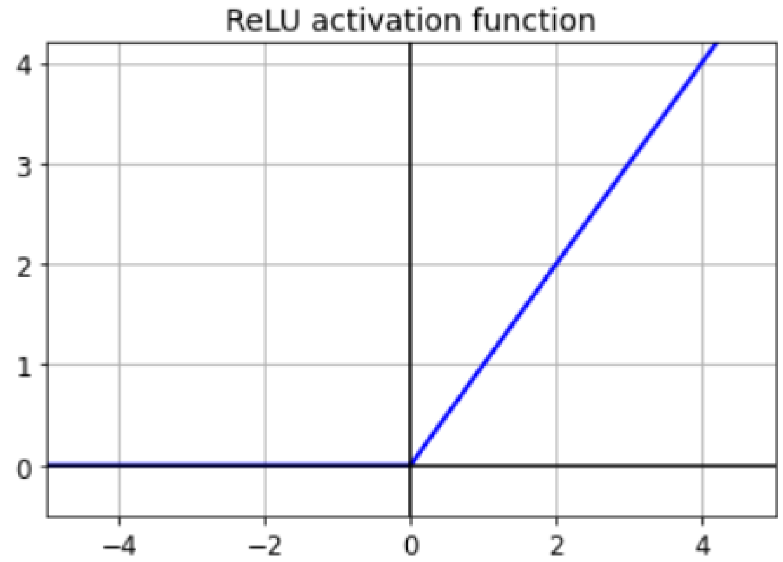


Table 11-1. Initialization parameters for each type of activation function

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

Non-Saturating Activation Functions


ReLU activation function

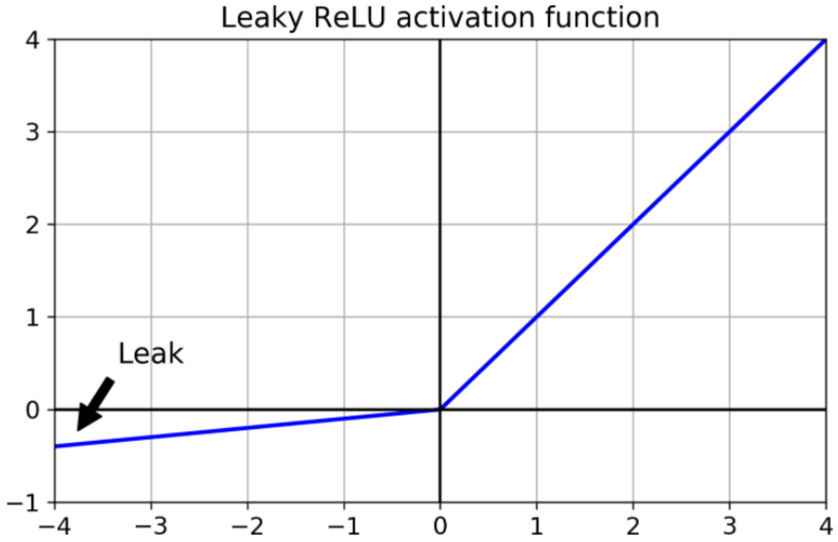
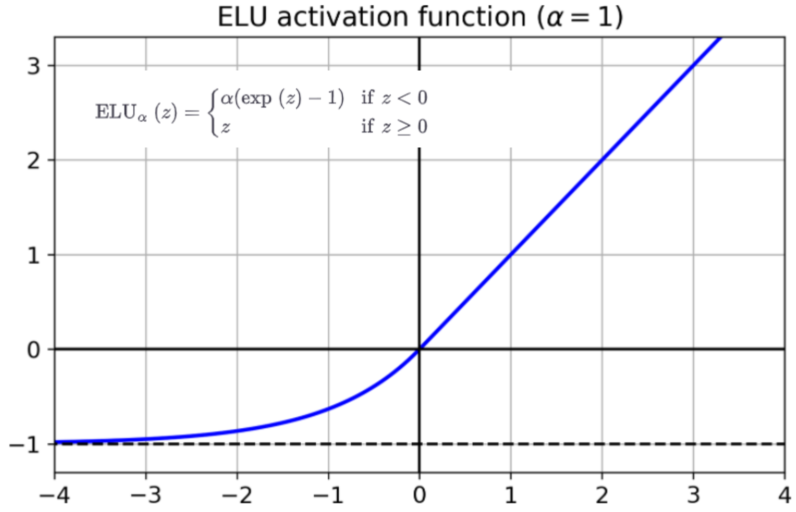
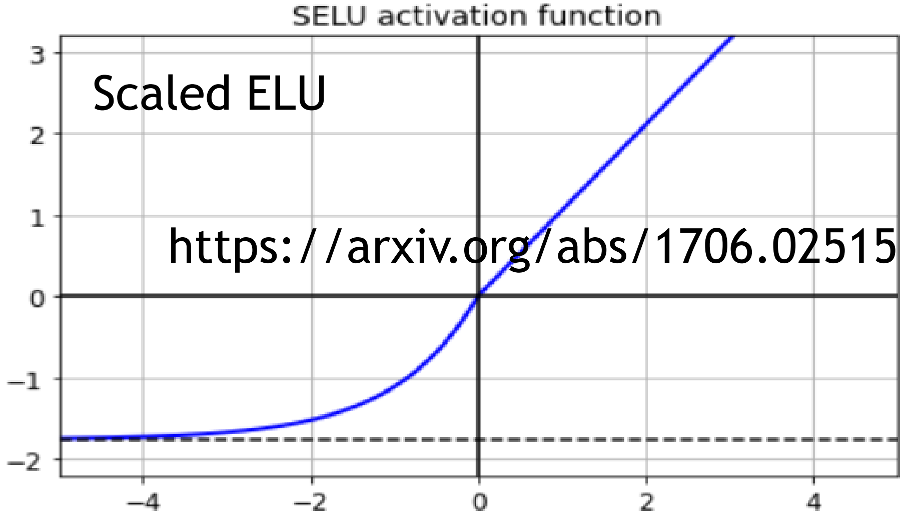

Leaky ReLU activation function

Figure 11-2. Leaky ReLU: like ReLU, but with a small slope for negative values


ELU activation function ($\alpha = 1$)

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

Figure 11-3. ELU activation function

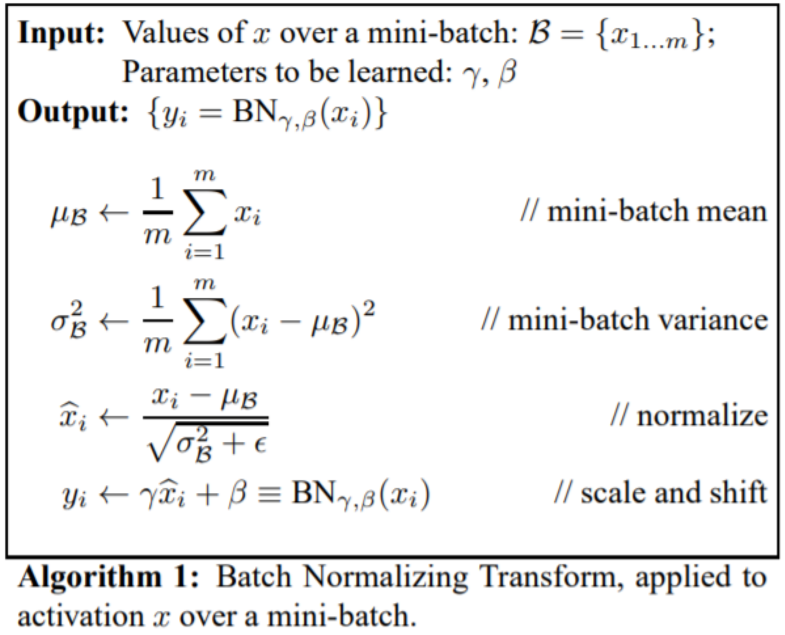

SELU activation function

Scaled ELU

https://arxiv.org/abs/1706.02515

Batch Normalization

- Ioffe and Szegedy, 2015
- Prevent Gradient Problems during Training
- Each Neuron (or Layer if CNN)
- Why does it work?
  - Internal Covariate Shift, Gradient Smoothing
  - Regularization



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

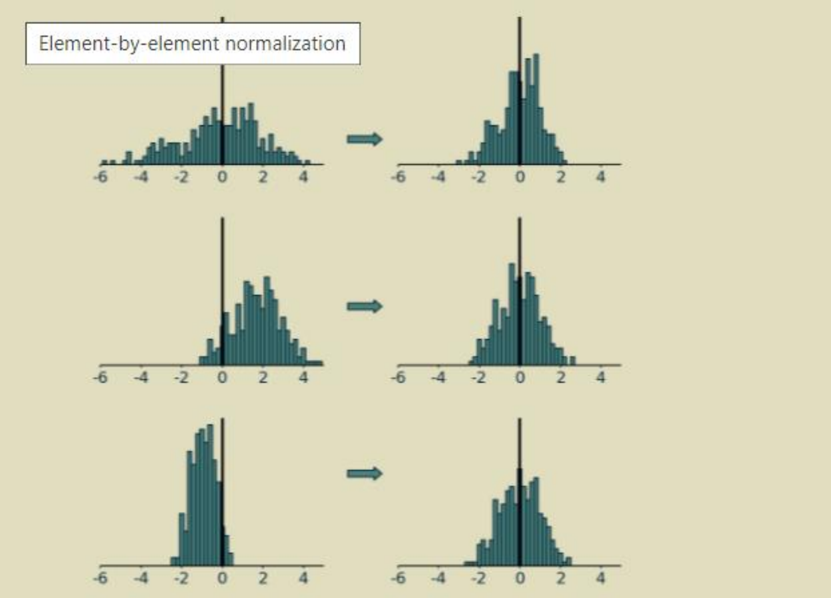
$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

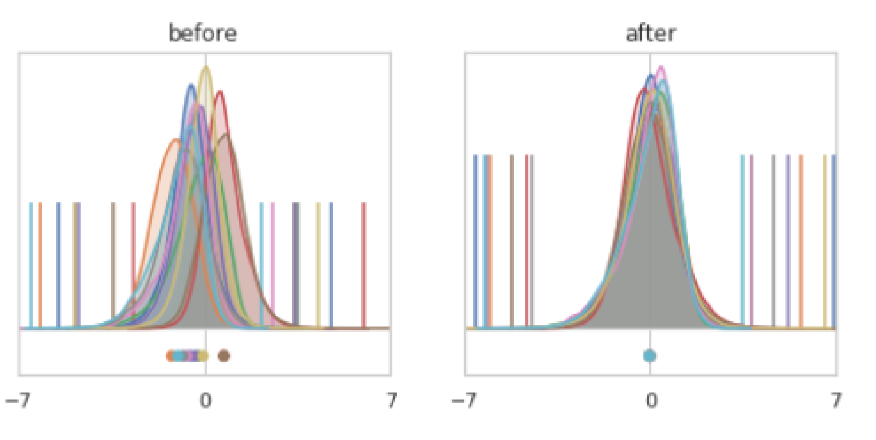
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.



Brandon Rohrer blog post



David C Page Colab Notebook

Implementing Batch Norm w/ Keras

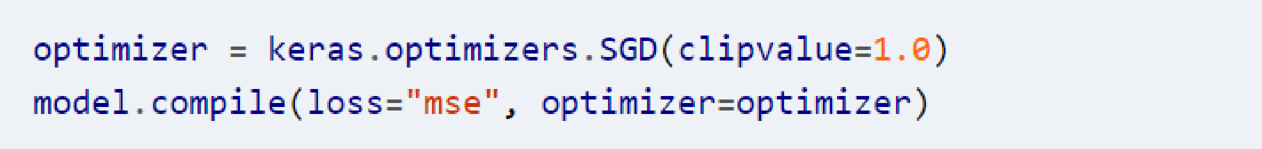
- Where to put BatchNorm
  - Typical: Separate Layer after Activation
  - Can do it before Activation instead (a la Ioffe and Szegedy)
- Parameters: Trainable vs Non-Trainable
- Inference vs Training
  - Training: separate mean/std calculation on each batch
  - Inference: Estimate dataset mean/std using momentum

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

- BN is Ubiquitous now
  - Omitted in diagrams / papers
  - But is it necessary? … (Fixed update initialization) https://arxiv.org/pdf/1901.09321.pdf

Gradient Clipping

- Exploding gradients don’t matter if you clip them during BackProp

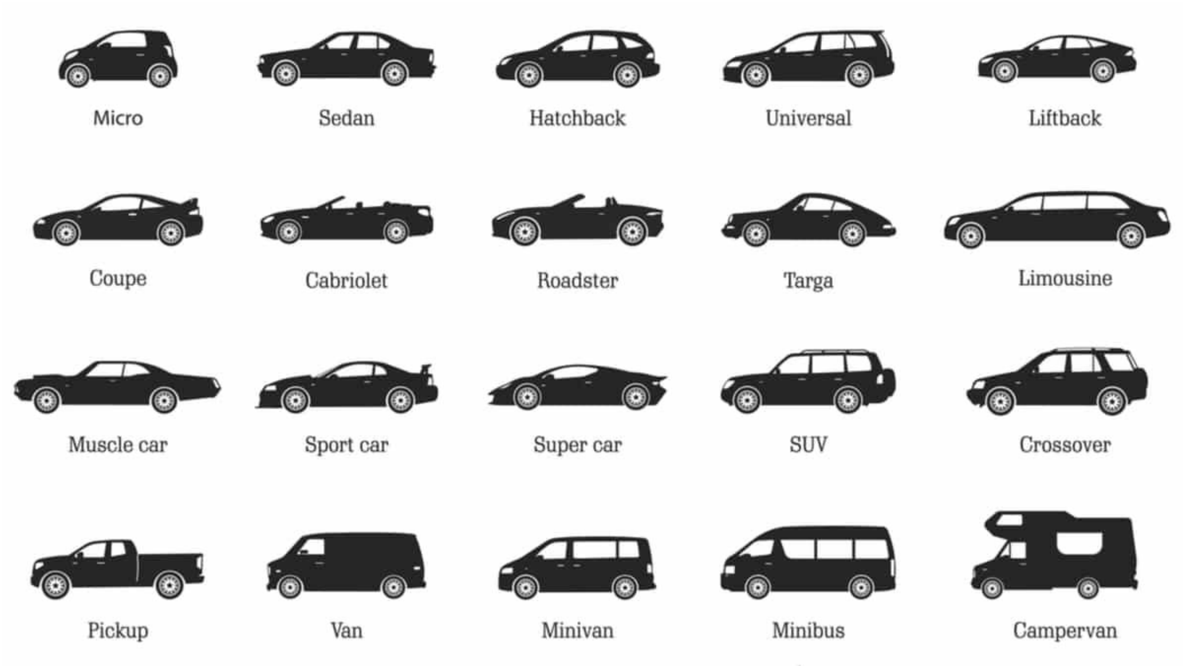


```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

- Used especially in RNNs
- Use clipnorm instead of clipvalue to preserve direction (L2 norm)

2. Reusing Pretrained Layers

- aka => Transfer Learning
- Freeze Lower Layers
- Retrain Upper Layers
- Replace Input/Output as Needed



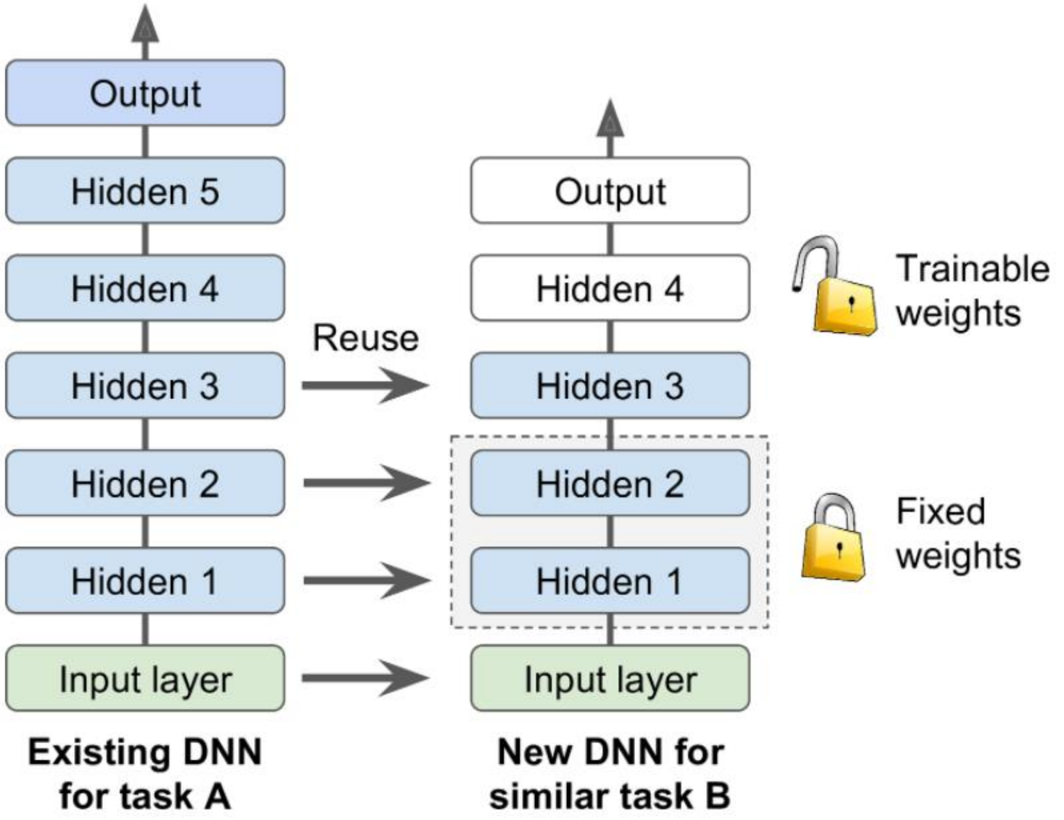
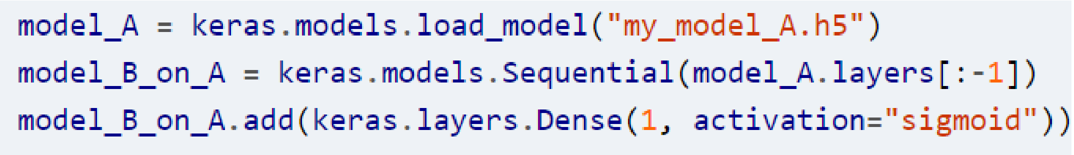


Figure 11-4. Reusing pretrained layers

Transfer Learning with Keras

- Fashion MNIST



```
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"))
```

- Clone avoids model_A overwrite
- One approach: train in stages.
  - Freeze middle layers at first
  - Train new/upper layers
  - Tweak middle after stabilization

- Geron Fashion MNIST example: Confession under Torture
- Transfer Learning works best with large/sparse models

Unsupervised Pretraining

- Data is cheap. Labels are expensive!
- Option: use Autoencoding
- Hinton (2006): Greedy Layer-Wise Pretraini

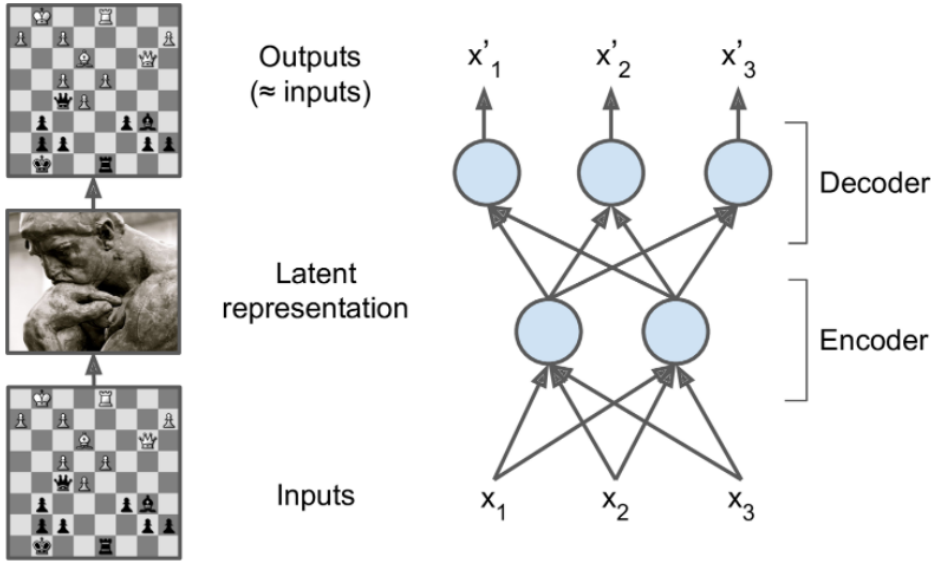


Figure 17-1. The chess memory experiment (left) and a simple autoencoder (right)

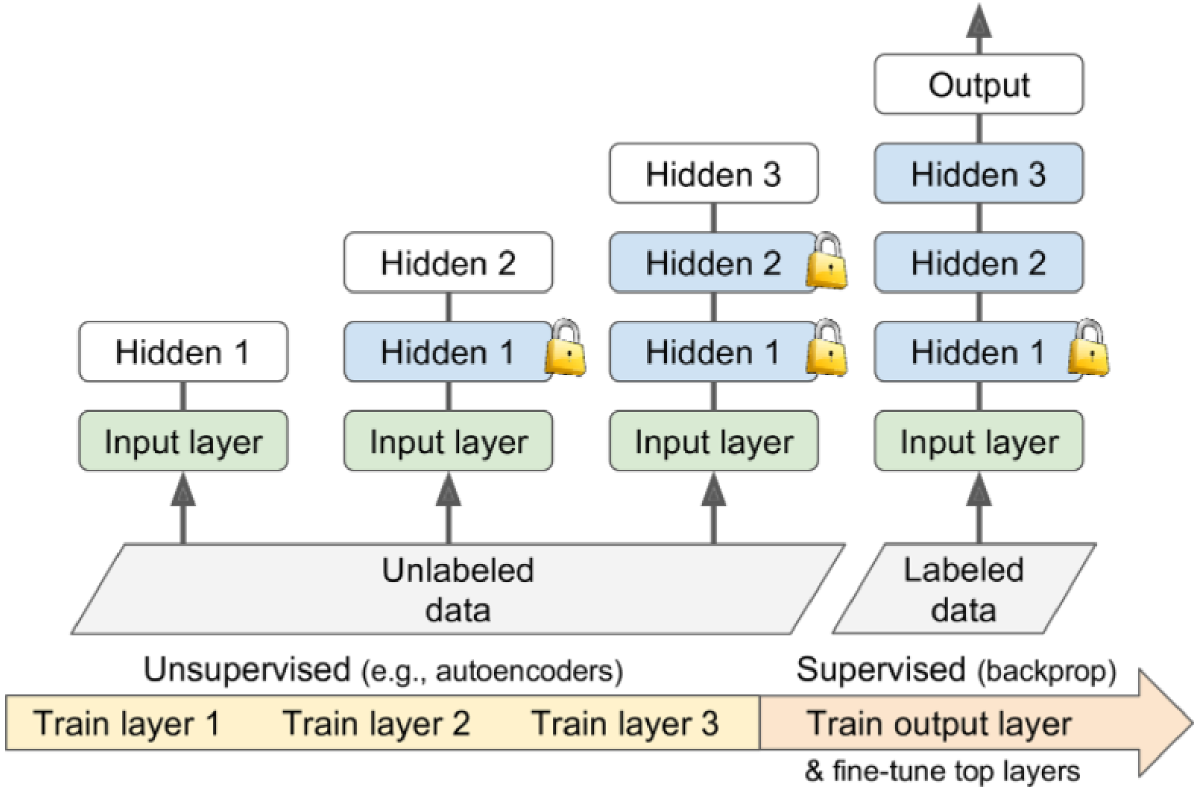


Figure 11-5. In unsupervised training, a model is trained on the unlabeled data (or on all the data) using an unsupervised learning technique, then it is fine-tuned for the final task on the labeled data using a supervised learning technique; the unsupervised part may train one layer at a time as shown here, or it may train the full model directly

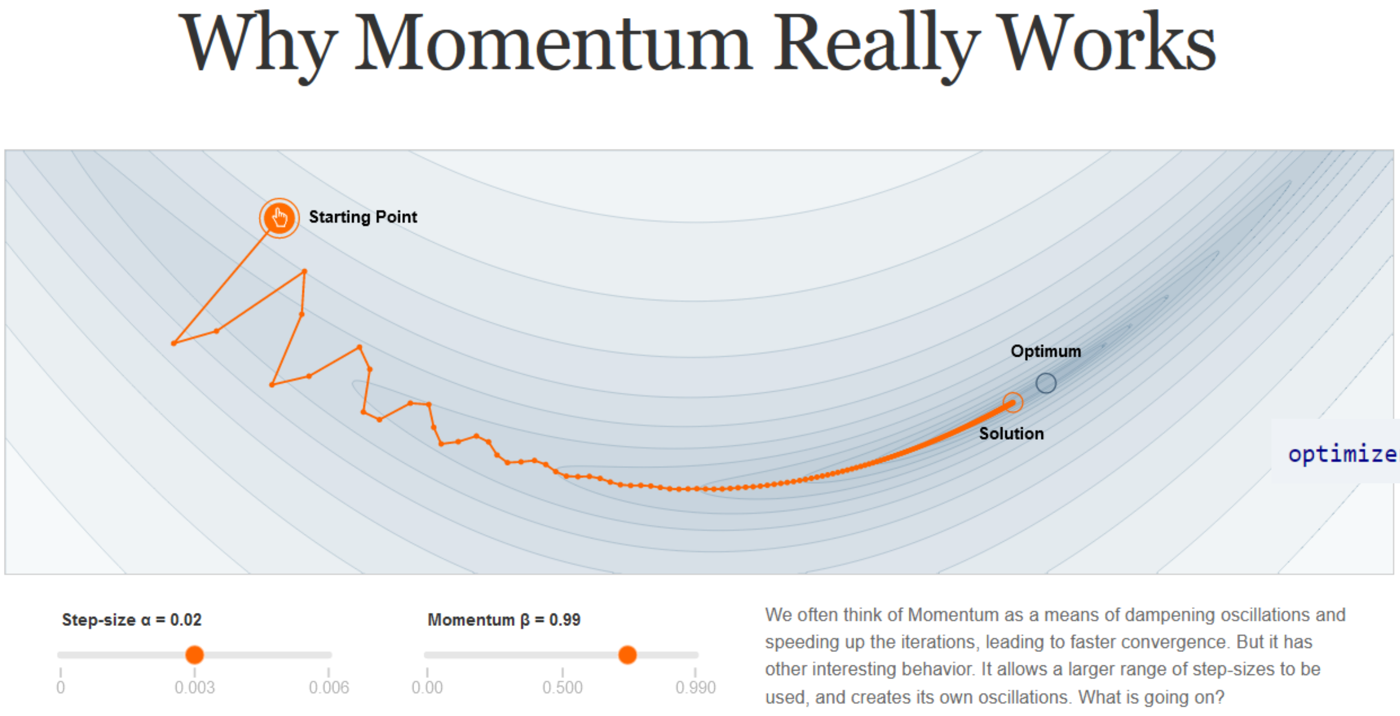
Pretraining on an Auxiliary Task

- Computer Vision Example
  - Face Classifier from very few images
  - Pull face data of random people from web
  - Train binary classifier for image pairs: is it the same person?

- NLP
  - Language models (e.g. BERT) are trained with masking
  - Can reuse latent representations for many NLP tasks

Faster Optimizers: Momentum

- Polyak 1964: Some methods of speeding up the convergence of iteration methods
- Gabriel Goh (2017): Interactive Visualization of Momentum

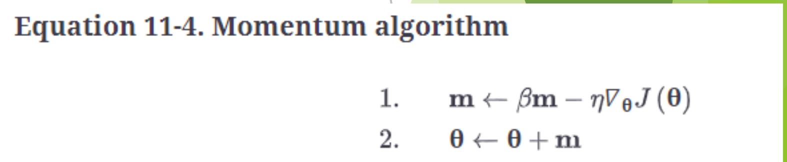


Why Momentum Really Works

Step-size α = 0.02

Momentum β = 0.99

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?



Equation 11-4. Momentum algorithm

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

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

Faster Optimizers: Nesterov Accelerated Gradient

- Yurii Nesterov (1983)

Equation 11-5. Nesterov Accelerated Gradient algorithm

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

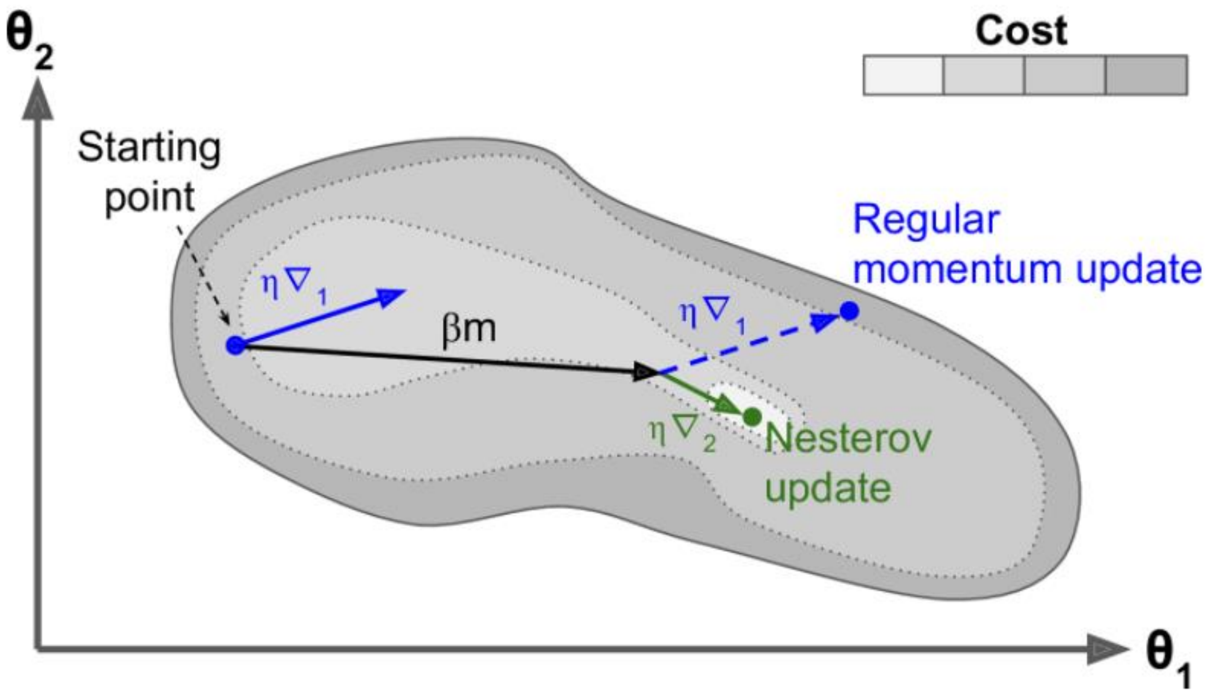
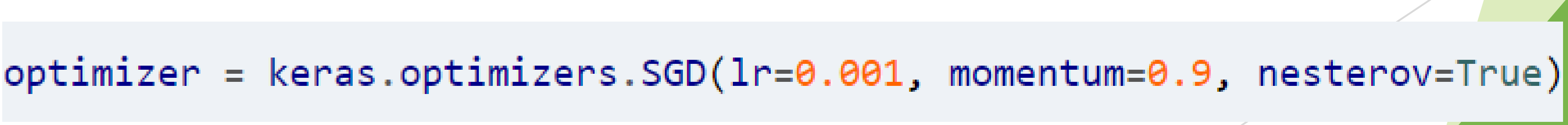


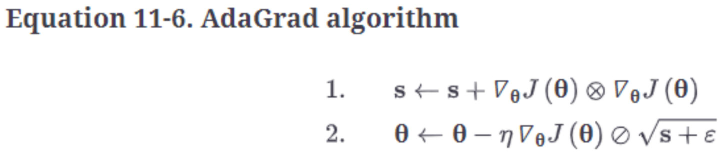
Figure 11-6. Regular versus Nesterov momentum optimization: the former applies the gradients computed before the momentum step, while the latter applies the gradients computed after



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

Faster Optimizers: AdaGrad

- Duchi et al: Adaptive Subgradient Methods for Online Learning and Stochastic Optimization (2011)
- Adaptive Learning Rate
  - Different across dimensions
- Good for quadratics
- Stops early in complex problems



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

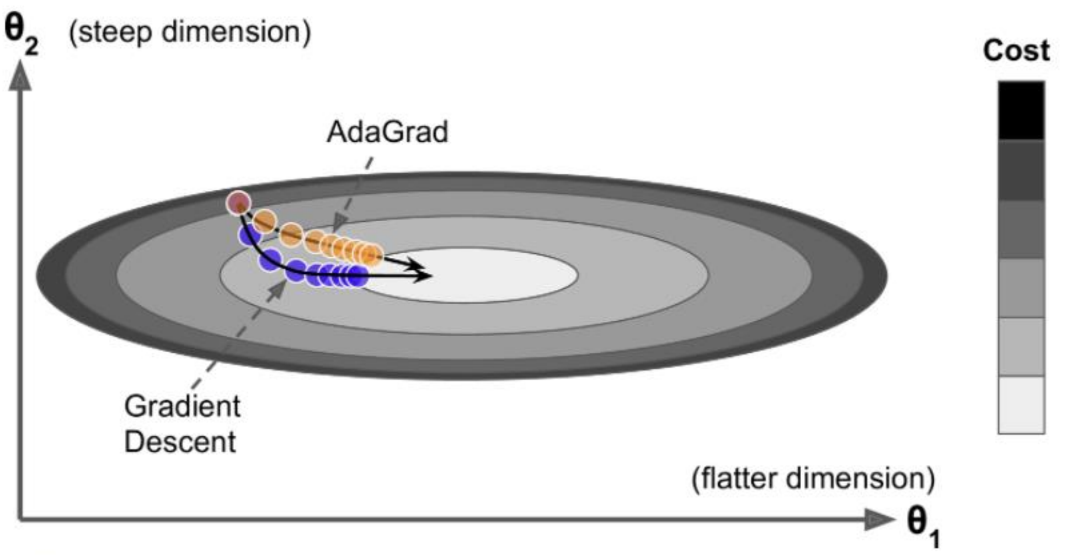
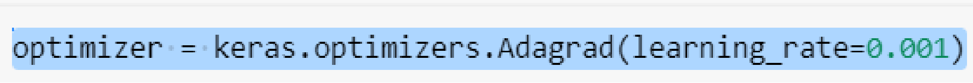


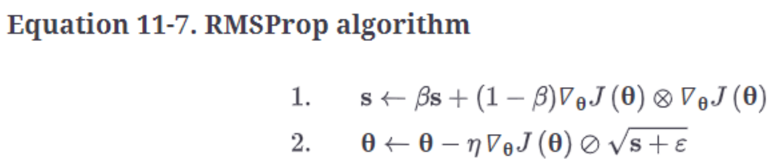
Figure 11-7. AdaGrad versus Gradient Descent: the former can correct its direction earlier to point to the optimum



```
optimizer = keras.optimizers.Adagrad(learning_rate=0.001)
```

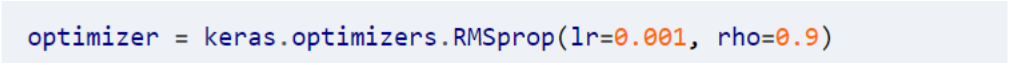
Faster Optimizers: RMSProp

- RMSProp improves AdaGrad
- Adds Exponential Decay to the first step
- Weights the recent squared gradients more strongly than early ones



Equation 11-7. RMSProp algorithm

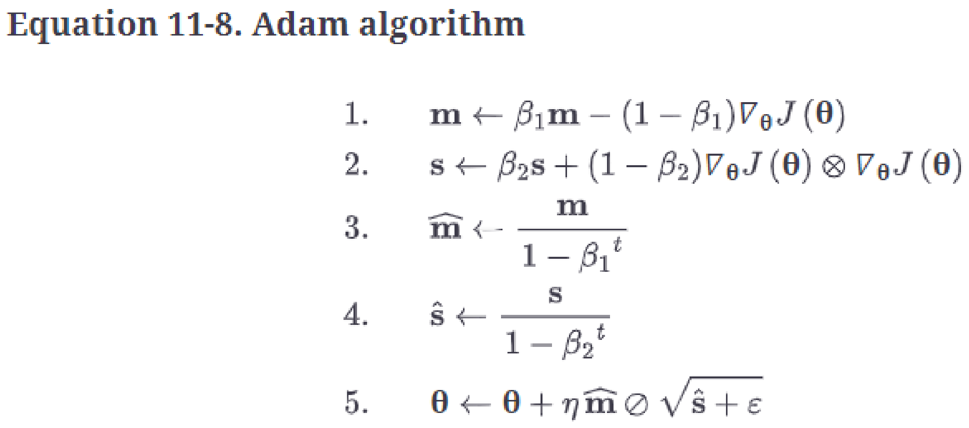
1. $\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s}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



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

Faster Optimizers: Adam and Nadam

- Adaptive Moment Estimation
- Combines Features from Momentum (1) and RMSProp (2 & 5)
- Steps 3&4 help to boost m & s at start of training



Equation 11-8. Adam algorithm

$$
\begin{aligned}
&1. \quad \mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\
&2. \quad \mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\
&3. \quad \widehat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - {\beta_1}^t} \\
&4. \quad \hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - {\beta_2}^t} \\
&5. \quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \widehat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \varepsilon}
\end{aligned}
$$

- More Optimizers:
  - Variants: AdaMax and Nadam
  - Hessian vs Jacobian Methods
  - [Tensorflow Model Optimization Toolkit]()

Learning Rate Scheduling

- Many options
  - Power Decay
  - Exponential
  - Piecewise Constant
  - Performance-based
  - 1cycle (Smith 2018)

Why Learning Rate Scheduling?

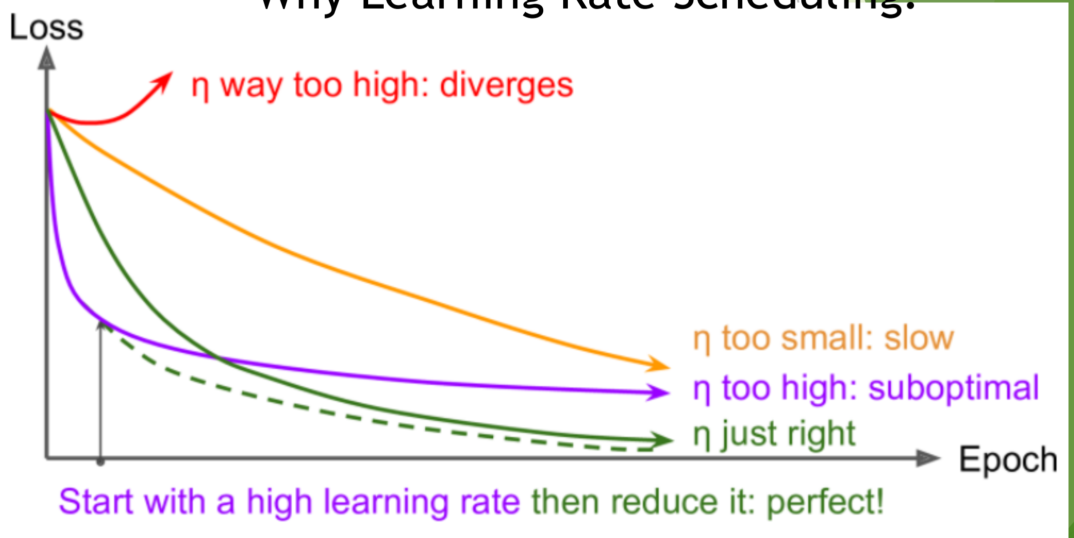
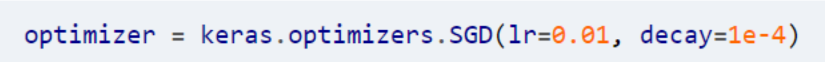
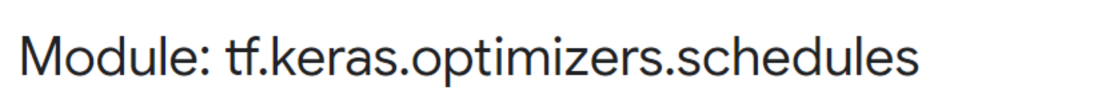


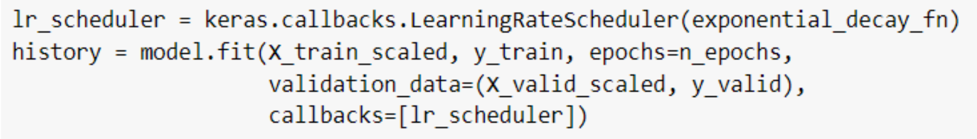
Figure 11-8. Learning curves for various learning rates η



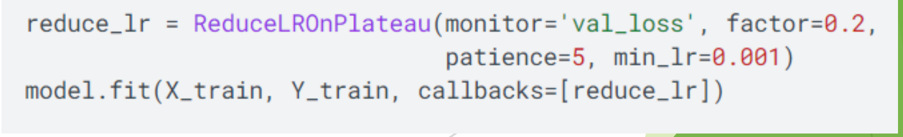
```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```



Module: tf.keras.optimizers.schedules

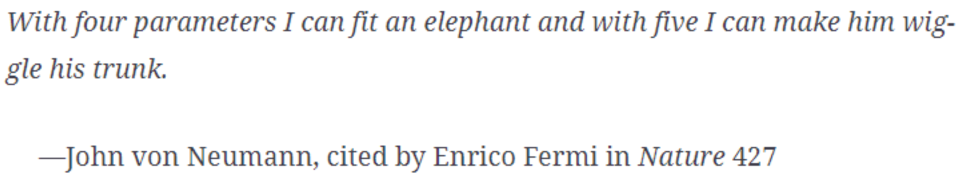


```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid),
                    callbacks=[lr_scheduler])
```



```
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2,
                              patience=5, min_lr=0.001)
model.fit(X_train, Y_train, callbacks=[reduce_lr])
```

Avoid Overfitting Through Regularization



> *With four parameters I can fit an elephant and with five I can make him wiggle his trunk.*
>
> —John von Neumann, cited by Enrico Fermi in *Nature* 427

- DNNs can have millions/billions of parameters
- How can we prevent overfitting?

- We've already seen some forms of regularization
  - Early Stopping
  - Batch Normalization

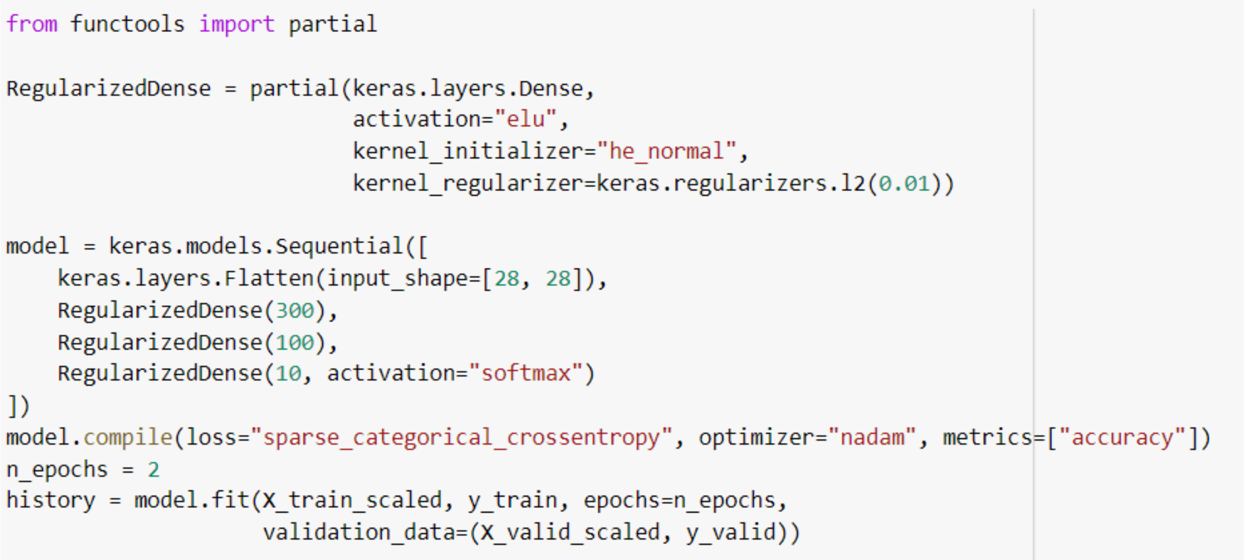
L1 and L2 Regularizatio

L2 at Google Devs ML Crash Course

$$\text{minimize}(\text{Loss}(\text{Data}|\text{Model}))$$

$$\text{minimize}(\text{Loss}(\text{Data}|\text{Model}) + \text{complexity}(\text{Model}))$$

$$L_2 \text{ regularization term} = ||\boldsymbol{w}||_2^2 = w_1^2 + w_2^2 + \ldots + w_n^2$$

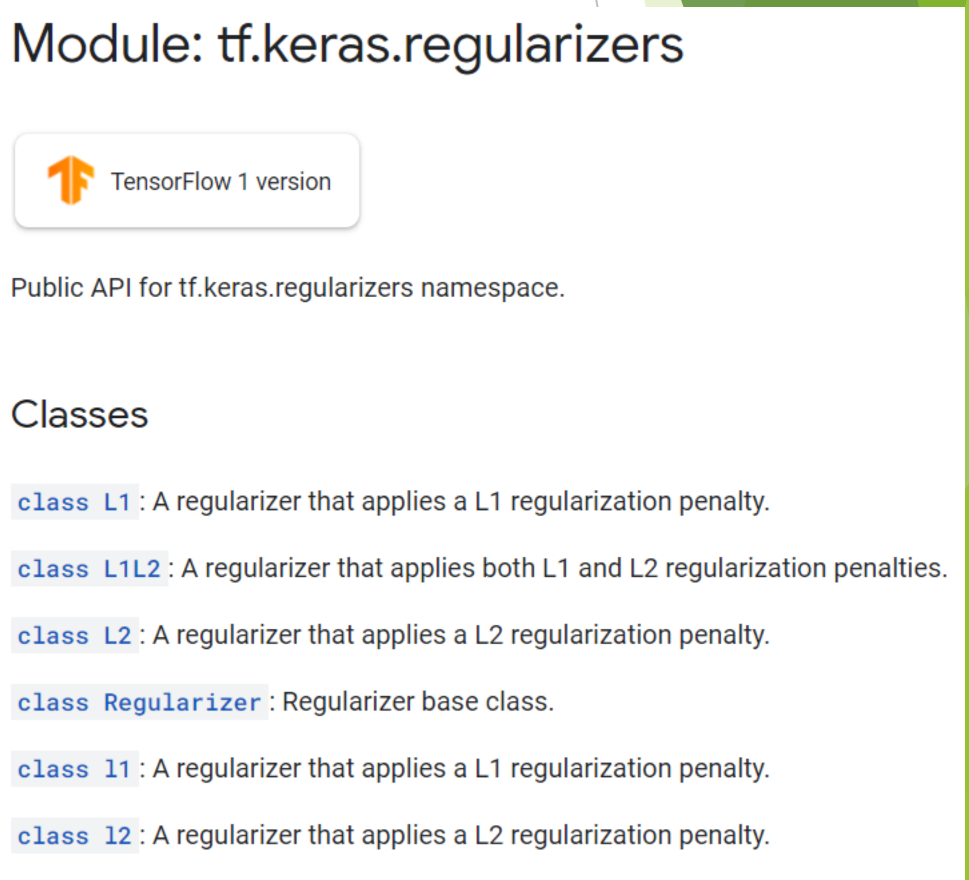


```
from functools import partial

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

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam", metrics=["accuracy"])
n_epochs = 2
history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid))
```

Regularizers in tf.keras



Module: tf.keras.regularizers

Public API for tf.keras.regularizers namespace.

Classes

`class L1` : A regularizer that applies a L1 regularization penalty.

`class L1L2` : A regularizer that applies both L1 and L2 regularization penalties.

`class L2` : A regularizer that applies a L2 regularization penalty.

`class Regularizer` : Regularizer base class.

`class l1` : A regularizer that applies a L1 regularization penalty.

`class l2` : A regularizer that applies a L2 regularization penalty.

Dropout

- Randomly break edges in the neural network
- Dropout probability *p*
- Analogy: Employee Attendance
- Implement yes/no Layer by Layer
- Test time
  - Activate all
  - Multiply activations by *(1-p)*
- Slower Convergence

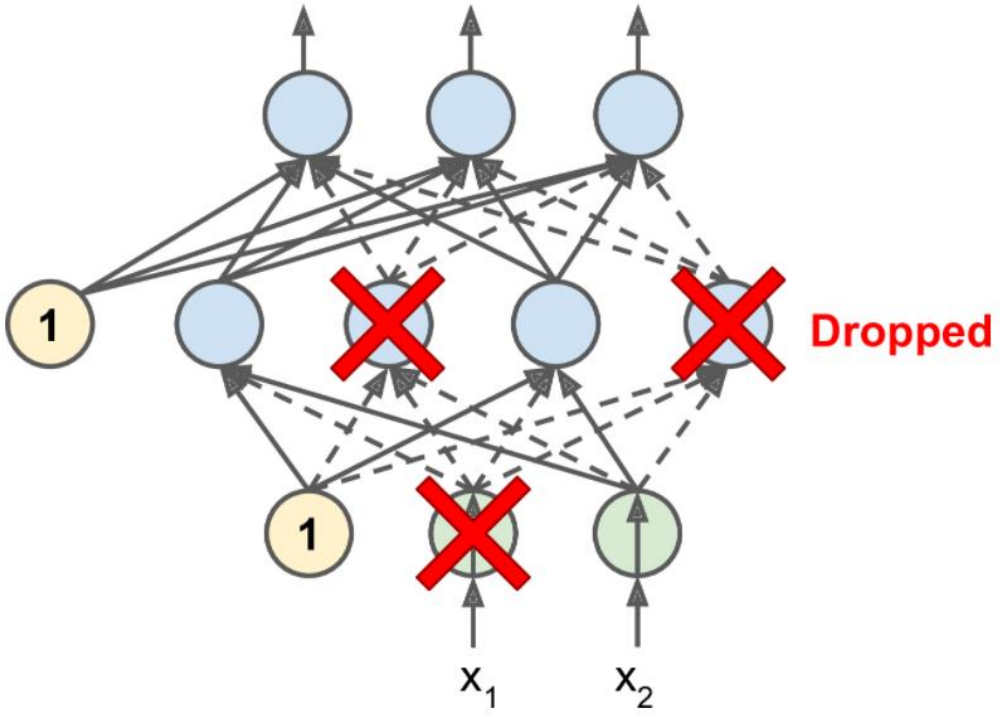
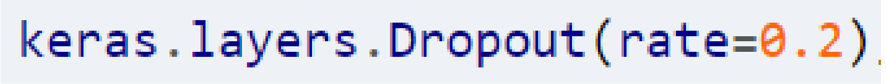
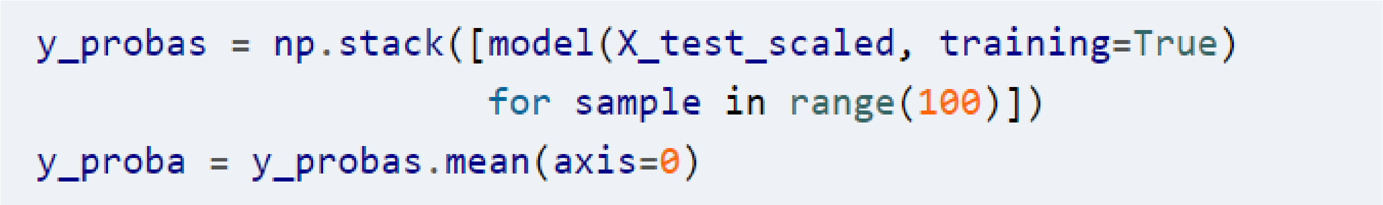


Figure 11-9. With dropout regularization, at each training iteration a random subset of all neurons in one or more layers—except the output layer—are “dropped out”; these neurons output 0 at this iteration (represented by the dashed arrows)



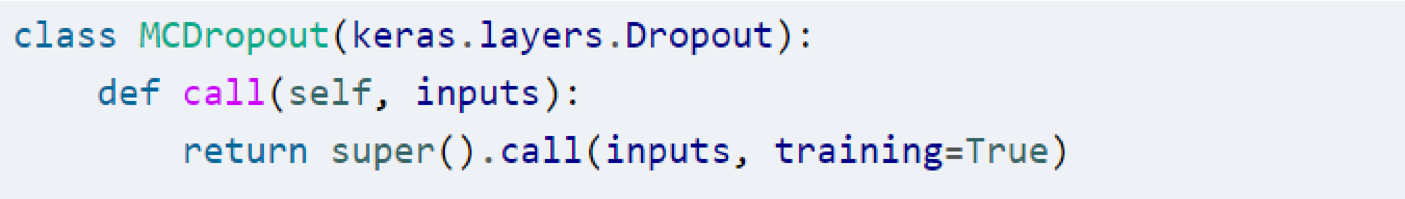
```
keras.layers.Dropout(rate=0.2)
```

Monte Carlo Dropout



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

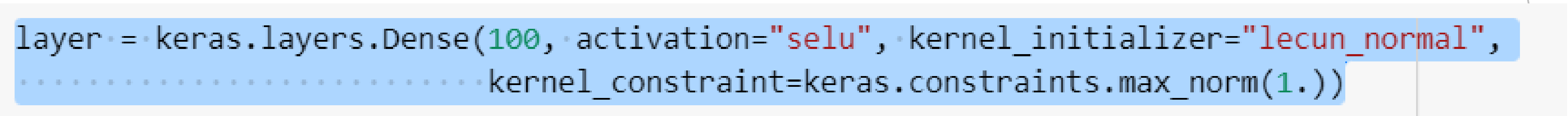
- Gal and Ghahramani, 2016: Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning
- training=True
  - Break connections during inference also!
- Average
- Ensemble of weak learners can outperform one strong learner



```
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

Max Norm Regularization

- Clipping size of weights, if necessary, after each training step



```
layer = keras.layers.Dense(100, activation="selu", kernel_initializer="lecun_normal",
                           kernel_constraint=keras.constraints.max_norm(1.))
```

Summary and Practical Guidelines

- Generally recommended starting hyperparameter configurations:

Default DNN configuration

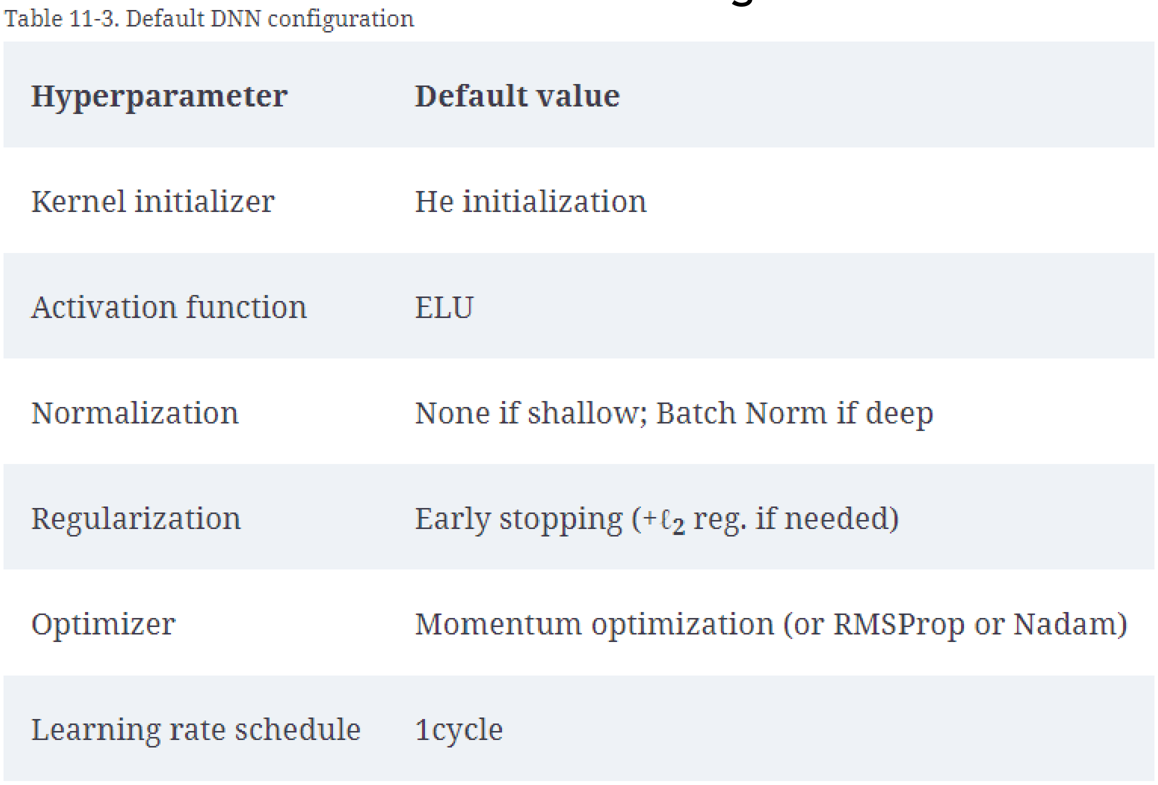


Table 11-3. Default DNN configuration

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ELU
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping (+ℓ_2 reg. if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

Self-normalizing NN configuration

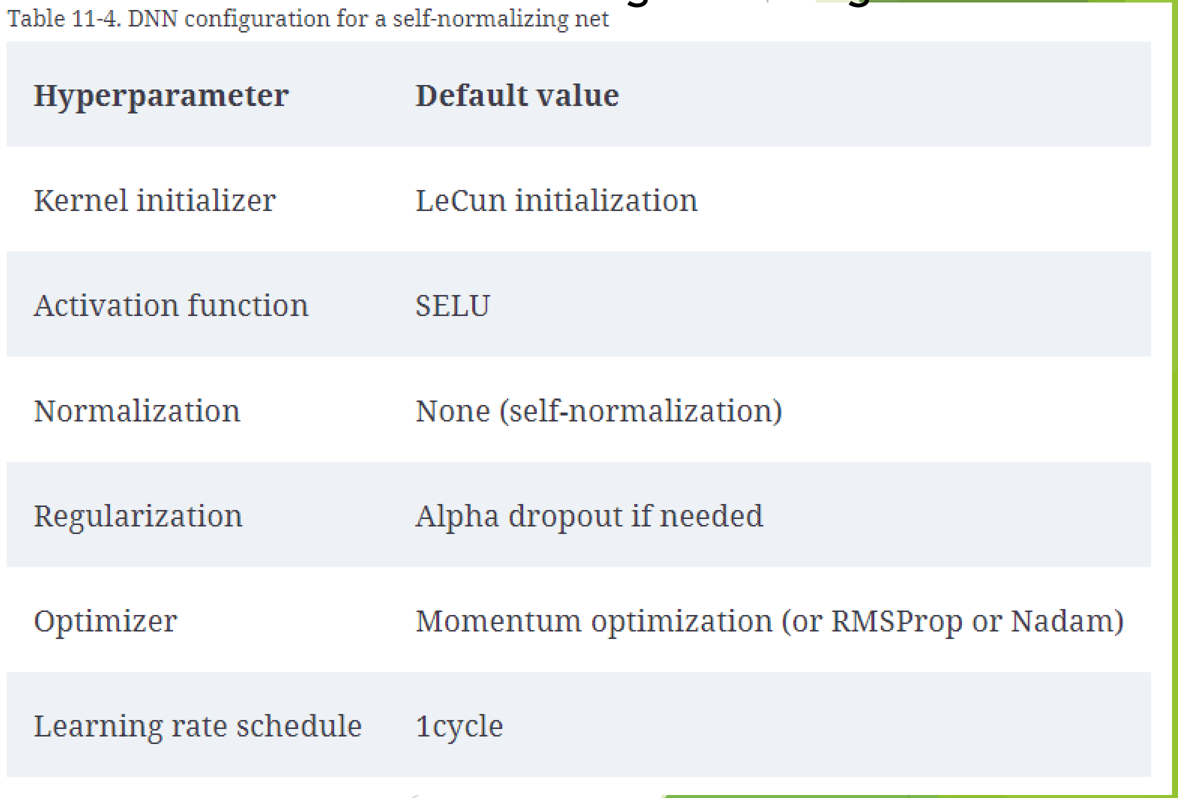


Table 11-4. DNN configuration for a self-normalizing net

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle