CITS5508 Machine Learning
Lectures for Semester 1, 2021
Lecture Week 10: Book Chapter 11

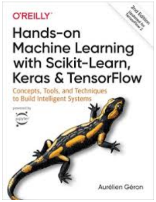
Dr Du Huynh (Unit Coordinator and Lecturer)
UWA

2021

Today

Chapter 11.

Hands-on Machine Learning with Scikit-Learn & TensorFlow



O'REILLY
Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow
Concepts, Tools, and Techniques to Build Intelligent Systems
Aurélien Géron

Training Deep Neural Networks

Here are the main topics we will briefly cover:

- Vanishing/Exploding Gradients Problems
  - Glorot and He initialization
  - Nonsaturating activation functions
  - Batch normalization
  - Gradient clipping
- Faster Optimizers
  - Momentum optimization
  - Nesterov Accelerated Gradient
  - RMSProp
  - Adam and Nadam optimization
  - Learning rate scheduling
- Avoiding overfitting
  - ℓ_1 and ℓ_2 regularization
  - Dropout

Problems in training DNNs

Training a DNN isn't a walk in the park. Here are some of the problems you could run into:

- the tricky *vanishing gradients* or *exploding gradients* problems;
- insufficient training data or insufficient labelled data to train the large network;
- training may be extremely slow;
- risk of overfitting the training data for a model with millions of parameters especially if there are not enough training instances or if they are too noisy.

Vanishing/Exploding Gradients Problems

- The *vanishing gradients* problem occurs when gradients become smaller and smaller as the algorithm progresses down to the lower layers. As a result, the Gradient Descent update leaves the lower layers' connection weights virtually unchanged, and training never converges to a good solution or the convergence is extremely slow. In other cases, we have the opposite problem, i.e., gradients become larger and larger, leading to the *exploding gradients* problem.
- This *vanishing gradients* problem was found[1] to be the combination of the popular logistic sigmoid activation function and the weight initialization technique that was most popular at the time.

[1] Xavier Glorot and Yoshua Bengio, "Understanding the Difficulty of Training Deep Feedforward Neural Networks," Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (2010): 249-256.

Vanishing/Exploding Gradients Problems (cont.)

- weight initialization techniques (usually drawn from $\mathcal{N}(0, 1)$);
- the variance of the outputs of each layer is much greater than the variance of its inputs;
- the logistic activation function has mean 0.5 rather than 0. Also, when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0.

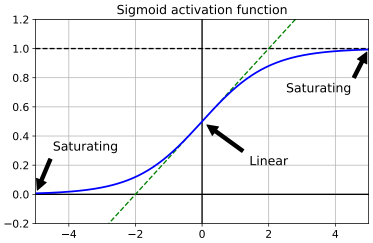


Figure 11-1. Logistic activation function saturation.

Vanishing/Exploding Gradients Problems (cont.)

Xavier or Glorot Initialization

Glorot and Bengio proposed a good compromise that has proven to work very well in practice: the connection weights between neurons must be initialized randomly as described below:

Glorot initialization (when using the logistic activation function)

- use the normal distribution with mean 0 and variance $\sigma^2 = \frac{1}{\text{fan}_{\text{avg}}}$, or
- use a uniform distribution over the interval $[-r, +r]$ with $r = \sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$

where[2] fan_{in} and fan_{out} denote the numbers of input and output units of a layer and $\text{fan}_{\text{avg}} = (\text{fan}_{\text{in}} + \text{fan}_{\text{out}})/2$

[2]Giuseppe Bonacorso, "Mastering Machine Learning Algorithms: Expert Techniques to Implement Popular Machine Learning Algorithms and Fine-tuning your Models", 2nd ed., 2020.

Vanishing/Exploding Gradients Problems (cont.)

He Initialization

He et al[3] provide similar weight initialization strategies for the ReLU activation function and its variants (including ELU):

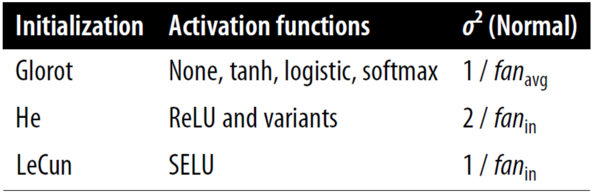
He initialization

Activation function	Uniform distr. $\mathcal{U}([-r,+r])$	Normal distr. $\mathcal{N}(0,\sigma^2)$
Logistic	$r = \sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$	$\sigma^2 = \frac{1}{\text{fan}_{\text{avg}}}$
Tanh	$r = 4\sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$	$\sigma^2 = \frac{16}{\text{fan}_{\text{avg}}}$
ReLU (& variants)	$r = \sqrt{2}\sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$	$\sigma^2 = \frac{2}{\text{fan}_{\text{avg}}}$

[3] Kaiming He et al., Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, CVPR 2015, 1026-1034.

Vanishing/Exploding Gradients Problems (cont.)

Summary of 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}

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

(He's σ^2 here is given in terms of fan_{in}. If $\text{fan}_{\text{in}} = \text{fan}_{\text{out}} = \text{fan}_{\text{avg}}$, then the formula for σ^2 is consistent with that on the previous slide)

Vanishing/Exploding Gradients Problems (cont.)

Weight initialization examples in Keras

Example: Use He's initialization strategy with the Normal distribution:

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

Example: Use the `VarianceScaling` initializer based on fan_{avg}:

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

Vanishing/Exploding Gradients Problems (cont.)

Nonsaturating Activation Functions

- The ReLU[4] activation function is commonly used because it does not saturate for positive values. Unfortunately, like the logistic activation function, it suffers from a problem known as the *dying ReLUs*: during training, some neurons effectively "die," meaning they stop outputting anything other than 0. This happens when the input to ReLU is negative.

[4]Recall that the ReLU (rectified linear unit) activation function is defined as: $ReLU(z) = \max(0, z)$.

Vanishing/Exploding Gradients Problems (cont.)

Nonsaturating Activation Functions

- To overcome this problem, variants of ReLU have been proposed, including
  - *leaky ReLU*, defined as $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$, where α is a small constant, e.g., $\alpha = 0.01$. Researchers found that a larger leak, e.g., $\alpha = 0.2$, tends to give better performance.

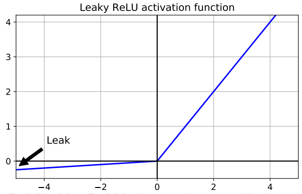


Figure 11-2. Leaky ReLU: like ReLU, but with a small slope for negative values.

  - *Randomized leaky ReLU* (RReLU), where α is picked randomly in a given range during training and is fixed to an average value during testing.

Vanishing/Exploding Gradients Problems (cont.)

Nonsaturating Activation Functions

- Variants of ReLU...
  - *Parametric leaky ReLU* (PReLU) – same as RReLU, except that α is learned during training together with the connection weights.
  - *Exponential linear unit*[5] (ELU) is defined as

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

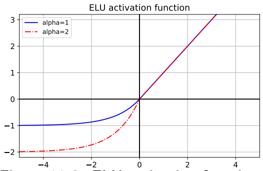


Figure 11-3. ELU activation function.

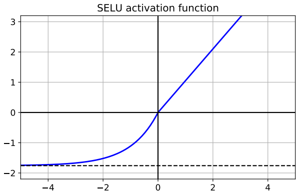
When $\alpha = 1$, the derivative at $z = 0$ is continuous also.

[5] Djork-Arné Clevert et al., "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)," International Conference on Learning Representations, 2016.

Vanishing/Exploding Gradients Problems (cont.)

Nonsaturating Activation Functions

- Variants of ReLU...
  - *Scaled ELU*[6] (SELU) – as the name suggests, this is a scaled version of ELU, i.e., $\mathrm{SELU}_{\alpha,\lambda}(z) = \lambda\,\mathrm{ELU}_{\alpha}(z)$, with an extra scaling parameter λ.



SELU activation function with $\alpha = 1.673$ and $\lambda = 1.051$.

SELU has the self-normalization property, i.e., preserving the mean and variance of each layer's output during training. Unfortunately, this property is easily broken when used together with ℓ_1 or ℓ_2, dropout, and non-sequential topology networks.

[6] Günter Klambauer et al., "Self-Normalizing Neural Networks," Proceedings of the 31st International Conference on Neural Information Processing Systems (2017): 972-981.

Vanishing/Exploding Gradients Problems (cont.)

Which activation function should you use?

- In general, it is recommended to follow this order: SELU $>$ ELU $>$ leaky ReLU (and its variants) $>$ ReLU $>$ tanh $>$ logistic.
- If the network's architecture prevents it from self-normalizing, then ELU may perform better than SELU (since SELU is not smooth at $z = 0$).
- If runtime latency is an issue, then you may prefer leaky ReLU.
- If you don't want to tweak yet another hyperparameter, you may use the default α values used by Keras (e.g., 0.3 for leaky ReLU).
- If you have spare time and computing power, you can try RReLU or PReLU.
- As ReLU is optimized in many libraries and hardware accelerators, if speed is your priority, ReLU might still be the best choice.

Vanishing/Exploding Gradients Problems (cont.)

Batch Normalization

Although using He initialization along with ELU (or any variant of ReLU) can mitigate the vanishing/exploding gradients problems at the beginning of training, it doesn't guarantee that they won't come back during training. To overcome these problems, the *batch normalization*[7] algorithm was proposed:

- An extra operation is inserted before or after the activation function in each hidden layer to normalize the mean to 0 and the variance to 1.
- This introduces two trainable parameters, γ and $\boldsymbol{\beta}$, that the training process must learn together with the connection weights.
- The mean and variance of each layer's outputs for entire training set are estimated using the moving average of the minibatches. They are used in the predictions on the testing set.

[7] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," ICML (2015): 448-456.

Vanishing/Exploding Gradients Problems (cont.)

Batch Normalization (cont.)

Summary of the algorithm:

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

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

$$\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$$

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

where $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B^2$ are the mean and variance vectors computed from the minibatch B; $\hat{\mathbf{x}}^{(i)}$ is the zero-centred and normalized input vector used internally in the BN layer; $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are the trainable scale and shift vectors (one parameter per input unit); and $\otimes$ represents element-wise multiplication.

Vanishing/Exploding Gradients Problems (cont.)

Batch Normalization (cont.)

- Batch Normalization (BN) has been found to also act like a regularizer, reducing the need for other regularization techniques (such as *dropout*);
- BN helps to remove the need for normalizing the input data (e.g., using `StandardScaler`);
- There is a runtime penalty though, due to the extra computations required at each layer; however, this is usually counterbalanced by the fact that convergence is much faster with BN, so it will take fewer epochs to reach the same performance.

Vanishing/Exploding Gradients Problems (cont.)

Implementing Batch Normalization with Keras

```
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")
])
```

Vanishing/Exploding Gradients Problems (cont.)

Implementing Batch Normalization with Keras

```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
batch_normalization (BatchNo (None, 784)               3136
_________________________________________________________________
dense (Dense)                (None, 300)               235500
_________________________________________________________________
batch_normalization_1 (Batch (None, 300)               1200
_________________________________________________________________
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
batch_normalization_2 (Batch (None, 100)               400
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

Vanishing/Exploding Gradients Problems (cont.)

Gradient Clipping

- Another popular technique to lessen the exploding gradients problem is to simply clip the gradients during backpropagation so that they never exceed some threshold (this is mostly useful for *recurrent neural networks* (RNNs) as BN is tricky to use in RNNs. This is called *Gradient Clipping*[8].
- For example,

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

  will clip every component of the gradient vector to a value between -1.0 and 1.0. An alternative is to use `clipnorm=1.0` if you want to retain the gradient vector orientation.

[8] Razvan Pascanu et al., "On the Difficulty of Training Recurrent Neural Networks," ICML (2013): 1310-1318.

Faster Optimzers

Huge speed boost comes from using a faster optimizer than the regular Gradient Descent optimizer. We will look at 4-5 of the most popular algorithms:

- *Momentum optimization*,
- *Nesterov accelerated gradient*,
- *RMSProp*, and
- *Adam and Nadam optimization*

Faster Optimzers (cont.)

Momentum optimization

- Recall that Gradient Descent simply updates the weights $\boldsymbol{\theta}$ as follows: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \nabla_{\theta} J(\boldsymbol{\theta})$. It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.
- In momentum optimization, a *momentum vector* $\mathbf{m}$ is used to keep track of gradients in the previous iterations. The updates of the weights involve using only $\mathbf{m}$, which contains information about the past momentum and the current error gradient. That is, the gradient is used as an acceleration term, rather than a speed term of the update. The algorithm can also help roll past local optima. It introduces a new hyperparameter $\beta < 1$ (usually set to 0.9). The update equations are:

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

  Note that past momentum vector $\mathbf{m}$ exponentially decays over iterations.

Faster Optimzers (cont.)

Momentum optimization (cont.)

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

Faster Optimzers (cont.)

Nesterov Accelerated Gradient

Proposed by Nesterov[9], this is a small variant of the momentum optimizer which measures the gradient of the loss function not at the local position $\boldsymbol{\theta}$ but slightly ahead in the direction of the momentum, at $\boldsymbol{\theta} + \beta\mathbf{m}$.

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

This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so the update ends up slightly closer to the optimum.

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

[9] Yurii Nesterov, "A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence O(1/k2)," Doklady AN USSR 269 (1983): 543-547.

Faster Optimzers (cont.)

RMSProp

In the *RMSProp* algorithm[10], instead of the moment vector $\mathbf{m}$, a vector $\mathbf{s}$ is used. The update equations are:

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

where $\beta < 1$ (default value 0.9 works well) and η are hyperparameters; ϵ is a small constant to avoid division by 0; operators $\otimes$ and $\oslash$ denote element-wise multiplication and division.

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

RMSProp was the preferred optimization algorithm of many researchers until Adam optimization came around.

[10]created by Tijmen Tieleman and Geoffrey Hinton in 2012.

Faster Optimzers

Adam Optimization

Adam (https://goo.gl/Un8Axa),[11] which stands for *adaptive moment estimation*, combines the ideas of Momentum optimization and RMSProp. So the algorithm involves both the $\mathbf{m}$ and $\mathbf{s}$ vectors. Instead of β, it has hyperparameters β_1 and β_2. The update equations are:

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. $\widehat{\mathbf{m}} \leftarrow \mathbf{m} / (1 - \beta_1^t)$
4. $\widehat{\mathbf{s}} \leftarrow \mathbf{s} / (1 - \beta_2^t)$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\, \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \epsilon}$

where steps 3 and 4 are normalization to help increase the magnitude $\mathbf{m}$ and $\mathbf{s}$ as both vectors are initialized at around zero (so may be a bit small); t is the iteration number.

[11] "Adam: A Method for Stochastic Optimization," D. Kingma, J. Ba (2015).

Faster Optimzers (cont.)

Adam Optimization (cont.)

Here is how to create an Adam optimizer using Keras:

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

The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999.

Faster Optimzers (cont.)

Nadam Optimization

Nadam optimization[12] is Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam. In Dozat's report, he compares many different optimizers on various tasks and finds that Nadam generally outperforms Adam but is sometimes outperformed by RMSProp.

[12] Timothy Dozat, "Incorporating Nesterov Momentum into Adam" (2016)

Faster Optimzers (cont.)

Learning rate scheduling

Finding a good learning rate is very important. If you set it too high, training may diverge; if you set it too low, training will eventually converge to the optimum, but it will take a very long time.

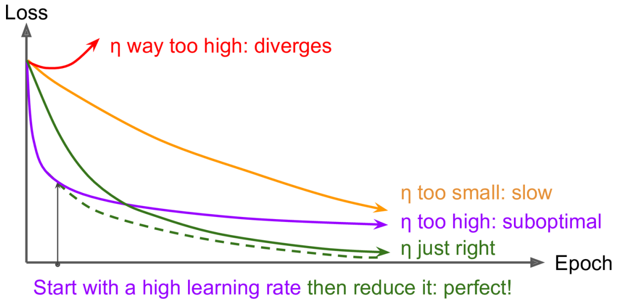


Figure 11-8. Learning curves for various learning rates η.

Faster Optimzers (cont.)

Learning rate scheduling (cont.)

If you start with a large learning rate and then reduce it once training stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate.

There are many different strategies to reduce the learning rate during training. It can also be beneficial to start with a low learning rate, increase it, then drop it again. These strategies are called *learning schedules*.

Faster Optimzers (cont.)

Learning rate scheduling (cont.)

Some commonly used learning schedules are listed below.

Power scheduling: Setting the learning rate as a function of the iteration number t, as follows:

$$\eta(t) = \frac{\eta_0}{(1 + t/s)^c}.$$

where η_0 is the initial learning rate, the power c (typically set to 1) and the step s are hyperparameters. After s steps, it is down to $\eta_0/2$. After s more steps, it is down to $\eta_0/3$, and so on.

Faster Optimzers (cont.)

Learning rate scheduling (cont.)

Exponential scheduling: Setting the learning rate to:

$$\eta(t) = \eta_0 0.1^{t/s}.$$

where the learning rate gradually drops by a factor of 10 every s steps.

Performance scheduling: Measure the validation error every N steps (just like for early stopping), and reduce the learning rate by a factor of λ when the error stops dropping.

Faster Optimzers (cont.)

Learning rate scheduling (cont.)

1Cycle scheduling[13] Contrary to the other approaches, 1Cycle starts by increasing η_0, growing linearly up to η_1 halfway through training, then it decreases the learning rate linearly down to η_0 again during the second half of training, finishing the last few epochs by dropping the learning rate down by several orders of magnitude (still linearly). The author reported that good validation accuracy was achieved with fewer training epochs on the CIFAR10 image dataset.

[13] Leslie N. Smith, "A Disciplined Approach to Neural Network Hyper-Parameters: Part 1Learning Rate, Batch Size, Momentum, and Weight Decay," arXiv preprint arXiv:1803.09820 (2018).

Faster Optimzers (cont.)

Learning rate scheduling (cont.)

Examples:

1. Implementing power scheduling in Keras:

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

The `decay` parameter is the same as $1/s$ in the formula. Keras assumes that $c = 1$.

2. Implementing your own exponential schedule:

```
def exponential_decay(lr0, s):
    def exp_decay_fn(epoch):
        return lr0 * 0.1**(epoch / s)
    return exp_decay_fn

my_exp_decay_fn = exponential_decay(lr0=0.01, s=20)

lr_scheduler = keras.callbacks.LearningRateScheduler(my_exp_decay_fn)

history = model.fit(X_train, y_train, ..., callbacks=[lr_scheduler])
```

Avoiding Overfitting

ℓ_1 and ℓ_2 Regularization

Just like you did for simple linear models, you can use ℓ_1 and ℓ_2 regularization to constrain a neural network's connection weights.

Example: To apply ℓ_2 regularization to a Keras layer's connection weights, using a regularization factor of 0.01:

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

Avoiding Overfitting (cont.)

Dropout

Dropout is one of the most popular regularization techniques for DNNs is arguably *dropout*.[14] It has proven to be highly successful: even the state-of-the-art neural networks got a 1–2% accuracy boost simply by adding dropout. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

[14] "Improving neural networks by preventing co-adaptation of feature detectors," G. Hinton et al. (2012).

Avoiding Overfitting (cont.)

Dropout (cont.)

It is a fairly simple algorithm:

- At each training step, every neuron (including the input neurons but excluding the output neurons) has a probability p of being **temporarily** "dropped out," meaning it will be entirely ignored during this training step; however, it may be active during the next step (see Figure 11-9).
- The hyperparameter p is called the *dropout rate*, and it is typically set to 50%.
- After training, neurons don't get dropped anymore. That is, at testing, you simply use the connection weights that have been learned from the training process.

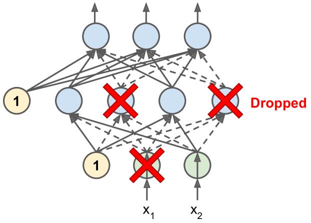


Figure 11-9. Dropout regularization

Avoiding Overfitting (cont.)

Dropout (cont.)

Example: The code below applies 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")
])
```

Summary

- Understand the vanishing and exploding gradients problems
- Understand the importance of properly initializing connection weights
- Know a few unsaturating activation functions
- Understand when to use batch normalization and how it works
- Know a few fast optimizers and their associated hyperparameters
- Understand learning rate scheduling.
- Understand ℓ_1 and ℓ_2 regularization
- Understand the concept of dropout

For next week

Work through the lab sheet and attend a supervised lab. The Unit Coordinator or a casual Teaching Assistant will be there to help.

Read up to Chapters 14 on Deep Computer Vision Using Convolutional Neural Networks

And that's all for this week's lecture.

Have a good week.