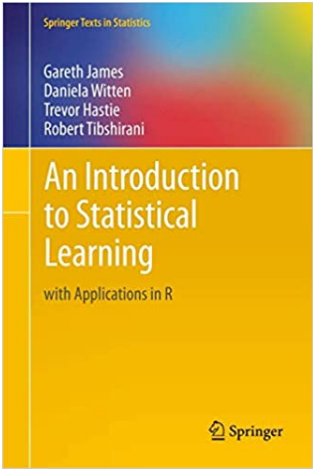
Deep Learning

Jui-Chung Yang[1]

May 2022

[1]National Taiwan University, jcyang1225@ntu.edu.tw

- James, Witten, Hastie, and Tibshirani (2021): Ch. 10, Deep Learning
  - Single Layer Neural Networks
  - Multilayer Neural Networks
  - Fitting a Neural Network


Springer Texts in Statistics

Gareth James
Daniela Witten
Trevor Hastie
Robert Tibshirani

An Introduction to Statistical Learning

with Applications in R

Springer

- Neural networks became popular in the 1980s.
  - Lots of successes, hype, and great conferences: NeurIPS, Snowbird.
- Then along came SVMs, Random Forests and Boosting in the 1990s.
  - Neural Networks took a back seat.
- Re-emerged around 2010 as Deep Learning. By 2020s very dominant and successful.
  - Part of success due to vast improvements in computing power, larger training sets, and software: Tensorflow and PyTorch.
- Much of the credit goes to three pioneers and their students: Yann LeCun, Geoffrey Hinton and Yoshua Bengio.
  - 2018 ACM Turing Award. https://awards.acm.org/about/2018-turing

Meanwhile, in Urbana-Champaign

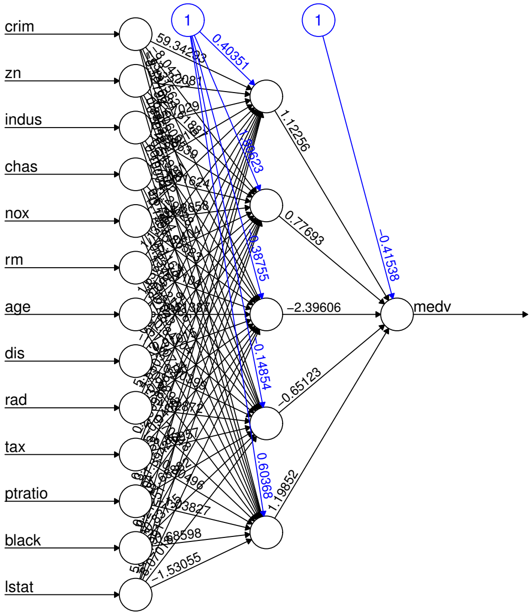
- C.-M. Kuan and H. White, "Some convergence results for learning in recurrent neural networks," in Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems, K. S. Narendra ed., pp. 103-109, New Haven: Yale University, 1990.
- C.-M. Kuan and K. Hornik, "Learning in a partially hard-wired recurrent network," Neural Network World , 1 , 39-45, 1991.
- C.-M. Kuan and K. Hornik, "Convergence of learning algorithms with constant learning rates," IEEE Transactions on Neural Networks , 2 , 484-489, 1991.
- K. Hornik and C.-M. Kuan, "Convergence analysis of local feature extraction algorithms," Neural Networks , 5 , 229-240, 1992.
- S. Piramuthu, C.-M. Kuan , and M. Shaw, "Learning algorithms for neural-net decision support," ORSA Journal on Computing , 5 , 361-373, 1993.
- C.-M. Kuan and H. White, "Artificial neural networks: An econometric perspective" with reply, Econometric Reviews , 13 , 1-91 and 139-143, 1994.
- C.-M. Kuan, K. Hornik, and H. White, "A convergence result for learning in recurrent neural networks," Neural Computation , 6 , 420-440, 1994.
- K. Hornik and C.-M. Kuan, "Gradient-based learning in recurrent networks," Neural Network World , 2/94 , 157-172, 1994.
- C.-M. Kuan, "A recurrent Newton algorithm and its convergence properties," IEEE Transactions on Neural Networks , 6 , 779-783, 1995.
- C.-M. Kuan and T. Liu, "Forecasting exchange rates using feedforward and recurrent networks," Journal of Applied Econometrics , 10 , 347-364, 1995.
- C.-M. Kuan, "Artificial neural networks," in New Palgrave Dictionary of Economics, S. N. Durlauf and L. E. Blume (eds.), Palgrave Macmillan, 2008.

Single (Hidden) Layer Feed-Forward Neural Network

$$\begin{aligned} f(\mathbf{x}_i) =& \beta_0 + \beta_1 A_{i1} + \beta_2 A_{i2} + \cdots + \beta_K A_{iK} \\ =& \beta_0 + \beta_1 h_1(\mathbf{x}_i) + \beta_2 h_2(\mathbf{x}_i) + \cdots + \beta_K h_K(\mathbf{x}_i) \\ =& \beta_0 + \sum_{k=1}^{K} \beta_k g_k \left(\omega_{k0} + \sum_{j=1}^{p} \omega_{kj} x_{ij} \right). \end{aligned}$$

- Fit by minimizing $L = \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i))^2$.

Network Diagram



- In a single (hidden) layer neural network, there is one **input layer**, one **hidden layer**, and one **output layer**.

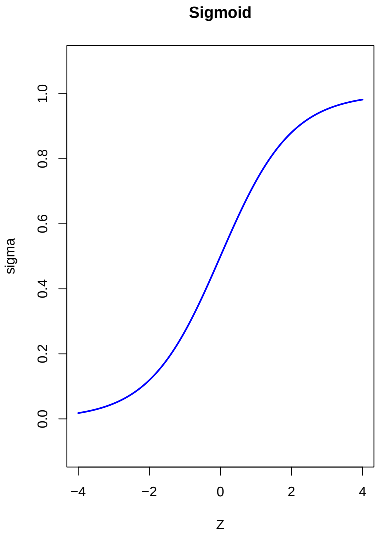
$$f(\mathbf{x}_i) = \beta_0 + \sum_{k=1}^{K} \beta_k g_k \left(\omega_{k0} + \sum_{j=1}^{p} \omega_{kj} x_{ij} \right).$$

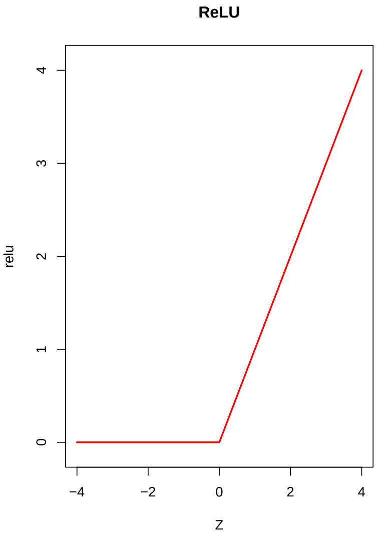
- p inputs, K hidden units (or *neurons*), and 1 output.
- $(p+1) \times K + (K+1)$ parameters.
  - In our example, 13 inputs, 5 hidden units, and 1 output.
  - $(13+1) \times 5 + (5+1) = 76$ parameters.
- $\{\omega_{kj}\}_{j=1}^{p}$ and $\{\beta_k\}_{k=1}^{K}$ are *weights*.
- $\{\omega_{k0}\}$ and β_0 are *bias parameters*, or *biases*.
- $g_k(\cdot)$ is known as the *activation function*.

- $g_k(\cdot)$ is known as the *activation function*.
  - As the neurons in the human brain, the idea was that each neuron in the network would be a simple binary on/off.
    - In practice, people usually consider smooth and differentiable compromises.
    - Activation functions in hidden layers are typically nonlinear, otherwise the model collapses to a linear model.
  - Popular are the **sigmoid** (logistic), the hyperbolic tangent, and **rectified linear**.

$$\begin{aligned}
\sigma(z) &= \frac{1}{1+\exp(-z)}, \\
\tanh(z) &= \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}, \\
g(z) &= z_+ = \max(0, z).
\end{aligned}$$

$$\sigma(z) = \frac{1}{1 + exp(-z)}, \text{ and } g(z) = z_+.$$





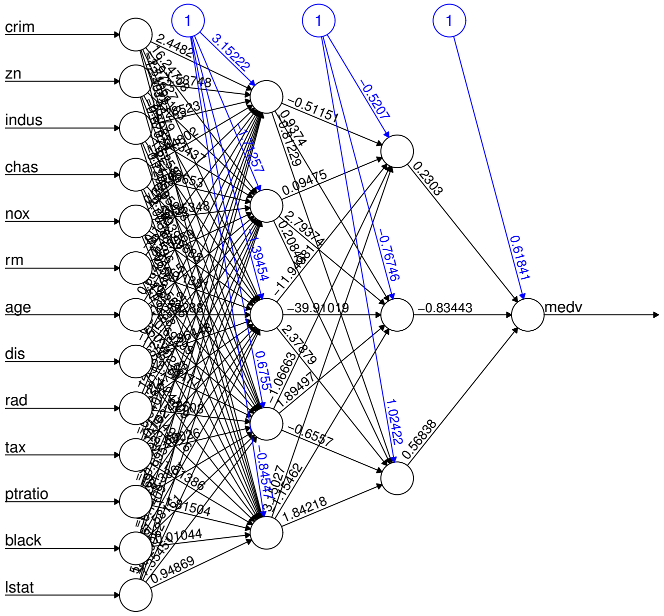
Multilayer Neural Network

- E.g., a neural network with two hidden layers:

$$A_{ik}^{(1)} = g_k\left(\omega_{k0}^{(1)} + \sum_{j=1}^{p} \omega_{kj}^{(1)} x_{ij}\right),\ A_{i\ell}^{(2)} = g_\ell\left(\omega_{\ell 0}^{(2)} + \sum_{k=1}^{K_1} \omega_{\ell k}^{(2)} A_{ik}^{(1)}\right),$$

$$f(\mathbf{x}_i) = \beta_0 + \sum_{\ell=1}^{K_2} \beta_\ell A_{i\ell}^{(2)}.$$

- p inputs, K_1 hidden units in the first hidden layer, K_2 hidden units in the second hidden layer, and 1 output.
- $(p+1) \times K_1 + (K_1+1) \times K_2 + (K_2+1)$ parameters.



crim
zn
indus
chas
nox
rm
age
dis
rad
tax
ptratio
black
lstat
1
1
1
3.15222
−0.5207
0.61841
−0.51151
0.09475
−39.91019
−0.83443
0.2303
0.56838
1.84218
−0.6557
−0.76746
1.02422
1.39454
0.6755
−0.84547
2.79374
2.37879
−1.89497
−1.06663
−11.94981
0.2303
medv

Universal Approximator

- What makes NN a useful econometric tool is its universal approximation property.
- A multi-layered NN with a large number of hidden units can well approximate a large class of functions.

- Hornik, K., Tinchcombe, M., White, H. (1989). Multilayer Feedforward Networks are Universal Approximators. Neural Networks. Vol. 2. Pergamon Press. pp. 359–366.
- Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. Neural Networks. 4 (2): 251–257.
- Lu, Z., Pu, H., Wang, F., Hu, Z., Wang, L. (2017). The Expressive Power of Neural Networks: A View from the Width". Advances in Neural Information Processing Systems. Curran Associates. 30: 6231–6239.
- Hanin, B., Sellke, M. (2019). Approximating Continuous Functions by ReLU Nets of Minimal Width. Mathematics. MDPI. 7 (10): 992.
- Kidger, P., Lyons, T. (2020). Universal Approximation with Deep Narrow Networks. Conference on Learning Theory.

Classification

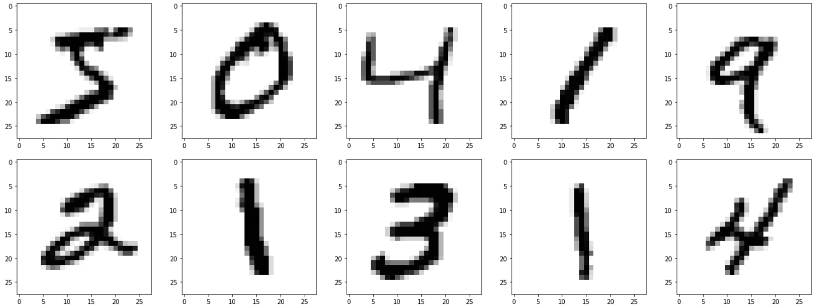
- For M-class classification, the number of output units is usually M.
- The final activation function is usually the **softmax** activation function:
$$f_m(\mathbf{x}_i) = \frac{\exp(A_m)}{\sum_{\ell=1}^{M} \exp(A_\ell)}.$$
- The softmax function computes a number (probability) between zero and one, and all M of them sum to one.
- To train this network, since the response is qualitative, we look for coefficient estimates that minimize the negative multinomial log-likelihood,

$$-\sum_{i=1}^{n} \sum_{\ell=1}^{M} y_i \ln \left(f_\ell(\mathbf{x}_i) \right)$$

Example: MNIST Digits

- Neural networks really cut their baby teeth on the optical character recognition (OCR) task.
- The `torchvision` package comes with a number of example datasets, including the `MNIST` digit data.
  - The *Modified National Institute of Standards and Technology* (`MNIST`) database (LeCun et al., 1998)
- Handwritten digits 28×28 grayscale images.
- 60K train, 10K test images.
- Features are the 784 pixel grayscale values $\in (0, 255)$.
- Labels are the digit class 0, 1, 2, ..., 9.
- Goal: build a classifier to predict the image class.

- 5, 0, 4, 1, 9, 2, 1, 3, 1, 4.



Methods	Error Rate
Human eyes (Simard et al., 1993)	2.5-2.0%
Linear models (LeCun et al., 1998)	12.0-7.6%
Neural nets (LeCun et al., 1998)	4.7-2.5%
Convolutional neural nets (LeCun et al., 1998)	1.7-0.7%
SVMs (LeCun et al., 1998)	1.1-0.8%
SVMs (DeCoste and Scholkopf, 2002)	0.68-0.56%
Neural nets (Ciresan et al., 2010)	0.35%
Convolutional neural nets (Ciresan et al., 2012)	0.23%

- Around 2000, outperformed by several other methods such as SVMs, NN were sidelined.
- Around 2010, NN were reincarnated, with **deep learning** as a flashier name.
  - SVMs and other methods were also known as *shallow learning* XD

- `mnist_dataset()` returns a `dataset()`.
  - A `dataset()` is a data structure implemented in `torch` allowing one to represent any dataset without making assumptions on where the data is stored and how the data is organized.

```r
train_ds_n <- mnist_dataset(root = ".", train = TRUE,
                            download = TRUE)
test_ds_n <- mnist_dataset(root = ".", train = FALSE,
                           download = TRUE)
str(train_ds_n[1])
str(test_ds_n[2])
length(train_ds_n)
length(test_ds_n)
```

- 60,000 images in the training data and 10,000 in the test data.
- Images are 28×28, and stored as matrix of pixels.

- Images are 28 $\times$ 28, and stored as matrix of pixels.
  - We need to transform each one into a vector.
- Neural networks are somewhat sensitive to the scale of the inputs.
  - Here inputs are eight-bit grayscale values between 0 and 255. We rescale to the unit interval.

```r
transform <- function(x) {
  x %>%
    torch_tensor() %>%
    torch_flatten() %>%
    torch_div(255)
}
train_ds <- mnist_dataset(
  root = ".",
  train = TRUE,
  download = TRUE,
  transform = transform
)
test_ds <- mnist_dataset(
  root = ".",
  train = FALSE,
  download = TRUE,
  transform = transform
)
```

- We define the intialize() and forward() methods of the nn_module().

```
modelnn <- nn_module(
  initialize = function() {
    self$linear1 <- nn_linear(in_features = 28*28, out_features = 256)
    self$linear2 <- nn_linear(in_features = 256, out_features = 128)
    self$linear3 <- nn_linear(in_features = 128, out_features = 10)

    self$drop1 <- nn_dropout(p = 0.4)
    self$drop2 <- nn_dropout(p = 0.3)

    self$activation <- nn_relu()
  },
  forward = function(x) {
    x %>%

      self$linear1() %>%
      self$activation() %>%
      self$drop1() %>%

      self$linear2() %>%
      self$activation() %>%
      self$drop2() %>%

      self$linear3()
  }
)
```

- In initialize we specify all layers that are used in the model.
  - For example, nn_linear(784, 256) defines a dense layer that goes from $28 \times 28 = 784$ input units to a hidden layer of 256 units.
- The model will have 3 of them, each one decreasing the number of output units.
- The last will have 10 output units, because each unit will be associated to a different class, and we have a 10-class classification problem.
- We also defined dropout layers using nn_dropout(). These will be used to perform dropout regularization.
- Finally we define the activation layer using nn_relu().
- In forward() we define the order in which these layers are called.
- We call them in blocks like (linear, activation, dropout), except for the last layer that does not use an activation function or dropout.

```
print(modelnn())
```

- Next, we add details to the model to specify the fitting algorithm.
- We fit the model by minimizing the cross-entropy function

$$-\sum_{i=1}^{n}\sum_{\ell=1}^{M} y_i \ln\left(f_\ell(\mathbf{x}_i)\right).$$

```
modelnn <- modelnn %>%
  setup(
    loss = nn_cross_entropy_loss(),
    optimizer = optim_rmsprop,
    metrics = list(luz_metric_accuracy())
  )
```

- Now we are ready to go. The final step is to supply training data, and fit the model.

```r
system.time(
   fitted <- modelnn %>%
      fit(
        data = train_ds,
        epochs = 5,
        valid_data = 0.2,
        dataloader_options = list(batch_size = 256),
        verbose = FALSE
      )
 )
```

- `valid_data = 0.2`. So training is actually performed on 80% of the 60,000 observations in the training set.
- I got a 4.8% error rate at home.

Fitting a Neural Network

- Despite the non-linearity, intuitively the neural net can be fit by a nonlinear least squares (NLS).

$$\frac{1}{n}\sum_{i=1}^{n} L\left(y_i, f(\mathbf{x}_i, \mathcal{W})\right).$$

  - $\mathcal{W}$ denotes the weights and bias parameters.
- However, this problem is difficult, with very *complex* and **non-convex** objective.
- Nowadays the estimation is usually via the **backpropagation** wit two general strategies.
  - *Slow Learning* using **gradient descent**.
  - *Regularization*.

Gradient Descent

- The idea of **gradient descent** is very simple.
- Suppose we represent all the parameters in one long vector θ. And our objective is to find a θ^* minimizing $R(\theta)$.

1. Start with a guess θ^0 for all the parameters in θ, and set $t = 0$.
2. Iterate until the objective $R(\theta)$ fails to decrease:
    - Find a vector δ that reflects a small change in θ, such that $\theta^{t+1} = \theta^t + \delta$ reduces the objective; i.e., such that $R(\theta^{t+1}) < R(\theta^t)$.
    - Set $t \leftarrow t + 1$.

- *Talk is cheap*. How do we find the δ?

- The **gradient** of $R(\theta)$, evaluated at some current value $\theta = \theta^m$, is the vector of partial derivatives at that point:

$$\nabla R(\theta^m) = \left. \frac{\partial R(\theta)}{\partial \theta} \right|_{\theta=\theta^m}.$$

- This gives the direction in θ-space in which $R(\theta)$ increases most rapidly.
- The idea of *gradient descent* is to move θ a little in the opposite direction (since we wish to go downhill):

$$\theta^{m+1} \leftarrow \theta^m - \rho \nabla R(\theta^m),$$

for a small enough but positive value of the learning rate ρ.

- For simplicity, let's first consider a single layer neural net.

$$f_\theta(\mathbf{x}_i) = \beta_0 + \sum_{k=1}^{K} \beta_k g_k \left(\omega_{k0} + \sum_{j=1}^{p} \omega_{kj} x_{ij} \right).$$

- Since $R(\theta) = \sum_{i=1}^{n} R_i(\theta) = \frac{1}{2} \sum_{i=1}^{n} (y_i - f_\theta(\mathbf{x}_i))^2$ is a sum, its gradient is also a sum over the n observations. So we will just examine one of these terms

$$\begin{aligned} R_i(\theta) &= \frac{1}{2} \left[y_i - \beta_0 - \sum_{k=1}^{K} \beta_k g_k (z_{ik}) \right]^2 \\ &= \frac{1}{2} \left[y_i - \beta_0 - \sum_{k=1}^{K} \beta_k g_k \left(\omega_{k0} + \sum_{j=1}^{p} \omega_{kj} x_{ij} \right) \right]^2. \end{aligned}$$

Backpropagation: Computing the Gradient

- First we take the derivative with respect to β_0 and β_k:

$$\frac{\partial R_i(\theta)}{\partial \beta_0} = -\left(y_i - f_\theta(\mathbf{x}_i)\right),$$
$$\frac{\partial R_i(\theta)}{\partial \beta_k} = -\left(y_i - f_\theta(\mathbf{x}_i)\right) \cdot g_k\left(z_{ik}\right).$$

- And now we take the derivative with respect to ω_{k0} and ω_{kj}.

$$\frac{\partial R_i(\theta)}{\partial \omega_{k0}} = \frac{\partial R_i(\theta)}{\partial g_k\left(z_{ik}\right)} \frac{\partial g_k\left(z_{ik}\right)}{\partial \omega_{k0}} = -\left(y_i - f_\theta(\mathbf{x}_i)\right) \cdot \beta_k \cdot g_k'\left(z_{ik}\right),$$
$$\frac{\partial R_i(\theta)}{\partial \omega_{kj}} = \frac{\partial R_i(\theta)}{\partial g_k\left(z_{ik}\right)} \frac{\partial g_k\left(z_{ik}\right)}{\partial \omega_{kj}} = -\left(y_i - f_\theta(\mathbf{x}_i)\right) \cdot \beta_k \cdot g_k'\left(z_{ik}\right) \cdot x_{ij}.$$

- $\theta^{m+1} \leftarrow \theta^m - \rho \nabla R(\theta^m)$
- The act of differentiation assigns a fraction of the residual $y_i - f_\theta(\mathbf{x}_i)$ to each of the parameters via the chain rule.
- The process is known as **backpropagation** in the neural network literature.

- Consider a K-layer NN:

$$\mathbf{z}_i^{(k)} = \mathbf{W}^{(k-1)}\mathbf{a}_i^{(k-1)},\ \mathbf{a}_i^{(k)} = g^{(k)}\left(\mathbf{z}_i^{(k)}\right).$$

  - $\mathbf{W}^{(k-1)}$ is a $p_k \times (p_{k-1}+1)$ matrix weights that go from layer $k-1$ to layer k.
  - $\mathbf{z}_i^{(k)}$ is the $p_k \times 1$ vector of linear transformation of $\mathbf{a}_i^{(k-1)}$.
  - $\mathbf{a}_i^{(k)}$ is the $(p_k+1) \times 1$ vector of activations at layer k.
    - $\mathbf{a}_i^{(0)} = \mathbf{x}_i$.
    - $p_K = 1$, $\mathbf{W}^{(K-1)} = [\beta_0\, \beta_1\, \cdots\, \beta_{p_{K-1}}]$, and $a_i^{(K)} = g^{(K)}\left(z_i^{(K)}\right) = z_i^{(K)}$.
- For output layer, $\nabla_{\mathbf{a}_i^{(K-1)}} R_i(\theta) = -\mathbf{W}^{(K-1)^\top}(y_i - f_\theta(\mathbf{x}_i))$, and

$$\nabla_{\mathbf{W}^{(K-1)}} R_i(\theta) = -\mathbf{a}_i^{(K-1)}(y_i - f_\theta(\mathbf{x}_i)).$$

- And for $k = 1, 2, \ldots, K-2$,

$$\nabla_{\mathbf{a}_i^{(k-1)}} R_i(\theta) = \nabla_{\mathbf{a}_i^{(k)}} R_i(\theta) \cdot \nabla_{\mathbf{z}_i^{(k)}} g^{(k)}\left(\mathbf{z}_i^{(k)}\right) \cdot \mathbf{W}^{(k-1)^\top}(y_i - f_\theta(\mathbf{x}_i)),$$

$$\nabla_{\mathbf{W}_i^{(k-1)}} R_i(\theta) = \nabla_{\mathbf{a}_i^{(k)}} R_i(\theta) \cdot \nabla_{\mathbf{z}_i^{(k)}} g^{(k)}\left(\mathbf{z}_i^{(k)}\right) \cdot \mathbf{a}_i^{(k-1)}(y_i - f_\theta(\mathbf{x}_i)).$$

Vanishing Gradient Problem

- Note that in the backpropagation we use $\nabla_{\mathbf{z}_i^{(k)}} g^{(k)} \left(\mathbf{z}_i^{(k)} \right)$.
- For logistic (and hyperbolic tangent) functions, $g'(z) \approx 0$ as z is away from zero.

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \Rightarrow \sigma'(z) = \frac{\exp(-z)}{(1 + \exp(-z))^2}.$$

- ReLU suffers less from the vanishing gradient problem.

$$g(z) = \begin{cases} 0, & \text{if } z < 0, \\ z, & \text{if } z \geq 0. \end{cases} \Rightarrow g'(z) = \begin{cases} 0, & \text{if } z < 0, \\ 1, & \text{if } z \geq 0. \end{cases}$$

- g' is very easy to computed and does not vanish when z is large.
  - For convenience, we define $g'(0)$ to be 1.

Stochastic Gradient Descent

- Gradient descent usually takes many steps to reach a local minimum.
- In practice, there are a number of approaches for accelerating the process.
- When n is large, instead of summing R_i over all n observations, **stochastic gradient descent** (SGD) randomly samples a **batch** of them each time we compute a gradient step.
- Besides the advantage in computation, the steps taken towards the golbal minimum also have oscillations that can help to get out of the local minima.
- An **epoch** of training means that all n training samples have been used in gradient steps, irrespective of how they have been grouped.

Regularization

- In the MNIST example, our model has 235,146 parameters in total!
  - Four times the number of training examples (60,000)!
- Regularization is essential here to avoid overfitting.
- Conventionally, people might do cross-validation to select the numbers of layers / neurons.
- Some considered ridge regularization.

$$\min R(\theta) + \lambda \|\theta_j\|_2^2.$$

Dropout Learning

- A relatively new and efficient form of regularization.
- Inspired by random forests, the idea is to randomly remove a fraction of the units in a layer when fitting the model.
- Figure 10.19 of James et al. (2021):

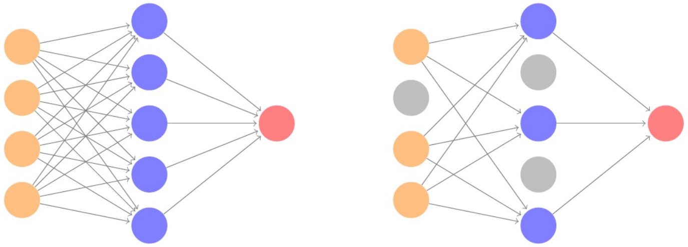


FIGURE 10.19. *Dropout Learning. Left: a fully connected network. Right: network with dropout in the input and hidden layer. The nodes in grey are selected at random, and ignored in an instance of training.*

https://jmlr.org/papers/v15/srivastava14a.html