Foundations of Statistical and Machine Learning for Actuaries - Artificial Neural Networks

Edward (Jed) Frees, University of Wisconsin - Madison
Andrés Villegas Ramirez, University of New South Wales

July 2025

Schedule

Day and Time	Presenter	Topics	Notebooks for Participant Activity
Monday Morning	Jed	Welcome and Foundations Hello to Google Colab	Auto Liability Claims
	Jed	Classical Regression Modeling	Medical Expenditures (MEPS)
Monday Afternoon	Andrés	Regularization, Resampling, Cross-Validation	Seattle House Sales Data
	Andrés	Classification	Victoria Road Crash Data
Tuesday Morning	Andrés	Trees, Boosting, Bagging	
	Jed	Big Data, Dimension Reduction and Non-Supervised Learning	Big Data, Dimension Reduction, and Non-Supervised Learning
Tuesday Afternoon	Jed	Neural Networks	Seattle House Prices Claim Counts
	Jed	Graphic Data Neural Networks	MNIST Digits Data
Tuesday 4 pm	Fei	Fei Huang Thoughts on Ethics	
Wednesday Morning	Jed	Recurrent Neural Networks, Text Data	Insurer Stock Returns
	Jed	Artificial Intelligence, Natural Language Processing, and ChatGPT	
Wednesday After Lunch	Dani	Dani Bauer Insights	
Wednesday Afternoon	Andrés	Applications and Wrap-Up	

Tuesday Afternoon 4A. Artificial Neural Networks

This module covers:

- a bit of neural net history
- the single layer feedforward network, including the model definition and approximation theorems
- fitting the model, illustrated by the Seattle House Price example
- the multiple layers feedforward network, including network tuning, backpropogation, and so on

During the lecture and lab, you can review:

- The *Seattle House Data* tutorial (**HousePriceRegression.ipynb**) shows how to fit a regression model using Python code. After examining the data, the first part fits a model using the sklearn package. Then, we use the keras package to fit an artificial neural network.
- The *Claim Counts* (**ClaimCounts.ipynb**) tutorial presents a study in the context of auto liability insurance focusing on claim counts. These data were introduced in the **AutoLiabilityClaims.ipynb** notebook.

A Short History of Artificial Neural Networks

At the risk of oversimplifying and for illustration purposes, some pivotal historical moments in deep learning include:

- 1962: Rosenblatt introduces the multilayer perceptron;
- 1967: Amari suggests training multilayer perceptrons with many layers via stochastic gradient descent;
- 1970: Linnainmaa publishes what is now known as backpropagation, the famous algorithm also known as "reverse mode of automatic differentiation" (it would take four decades until it became widely accepted);
- 1974–1980: first major "AI winter" (i.e., period of reduced funding and interest in AI);
- 1987–1993: second major "AI winter"

{ *Source:* Portfolio Optimization: Theory and Applications, 2025, by Daniel P. Palomar }

More History of ANNs

- 1997: LSTM networks introduced (Hochreiter & Schmidhuber, 1997);
- 1998: CNN networks established (LeCun et al., 1998);
- 2010: “AI spring” starts;
- 2012: AlexNet network achieves an error of 15.3
- 2014: GAN networks established and gained popularity for generating data;
- 2015: AlphaGo by DeepMind beats a professional Go player;
- 2016: Google Translate (originally deployed in 2006) switches to a neural machine translation engine;
- 2017: AlphaZero by DeepMind achieves superhuman level of play in the games of chess, Shogi, and Go;

Yet More History of ANNs

- 2017: Transformer architecture is proposed based on the self-attention mechanism, which would then become the de facto architecture for most of the subsequent DL systems (Vaswani et al., 2017);
- 2018: GPT-1 (Generative Pre-Trained Transformer) for natural language processing with 117 million parameters, starting a series of advances in the so-called large language models (LLMs);
- 2019: GPT-2 with 1.5 billion parameters;
- 2020: GPT-3 with 175 billion parameters;
- 2022: ChatGPT: a popular chatbot built on GPT-3, astonished the general public, sparking numerous discussions and initiatives centered on AI safety;
- 2023: GPT-4 with ca. 1 trillion parameters (OpenAI, 2023), which allegedly already shows some sparks of artificial general intelligence (AGI) (Bubeck et al., 2023).

{ *Source:* Portfolio Optimization: Theory and Applications, 2025, by Daniel P. Palomar } 7 / 46

Feedforward (Single Layer) Artificial Neural Network

Defining the model

- Consider an input vector of p variables $\mathbf{X} = (X_1, X_2, \dots, X_p)$.
  - We seek to build a nonlinear function $f(\mathbf{X})$ to predict the response Y.
- We build a layer that is intermediate between the input and predictions (output) called a *hidden layer*.
  - For $k = 1, \dots, K$, define

$$h_k(\mathbf{X}) = \sigma\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right)$$

  - Here, $\sigma(\cdot)$ is call the *activation function* and w_{kj} are weights.

- With this hidden layer, we can build predictions of the form

$$\hat{\mathbf{Y}} = \beta_0 + \sum_{k=1}^{K} \beta_k \; h_k(\mathbf{X}).$$

  and use

$$f(X) = \begin{cases} I(\hat{\mathbf{Y}} > threshold) & \text{binary classification problems each output unit implements a threshold function} \\ \hat{\mathbf{Y}} & \text{regression uses an identity function} \\ \frac{\exp(\hat{\mathbf{Y}})}{\sum \exp(\hat{\mathbf{Y}})} & \text{for multiclass classification problems, one can use a softmax function} \end{cases}$$

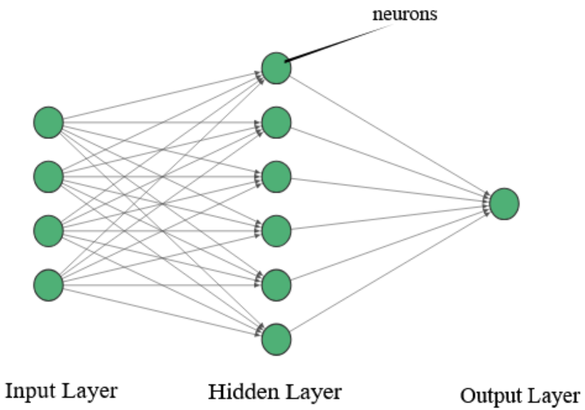
- Let's express this using matrix notation. We use inputs the vector of inputs $\mathbf{X} = (1, X_1, X_2, \ldots, X_p)$ and weights to define

$$\boxed{\begin{aligned} h_k = h_k(\mathbf{X}) &= \sigma\left(\mathbf{w}_k'\mathbf{X}\right) \\[2ex] f(\mathbf{X}) &= (\beta_0, \ldots, \beta_K)(1, h_1, \ldots, h_K)' \\ &= \beta'\mathbf{h} \end{aligned}}$$

the hidden units ("neurons") and the outputs.

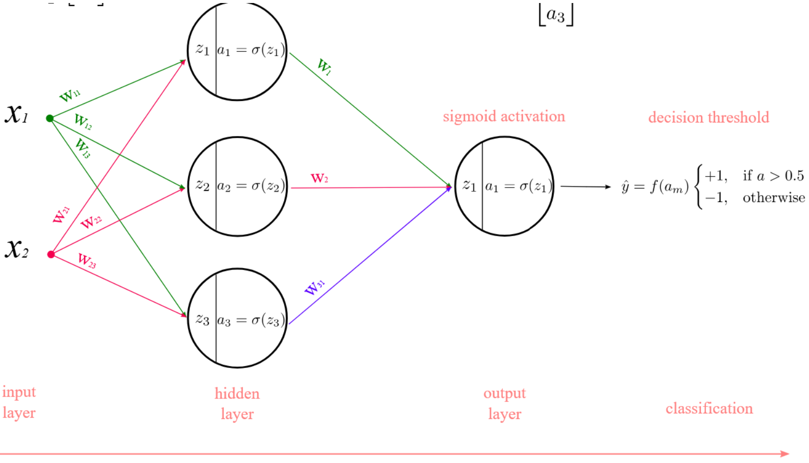
Graphical Model Summary. It is common to display these equations graphically.

This figure was built using https://alexlenail.me/NN-SVG/AlexNet.html.



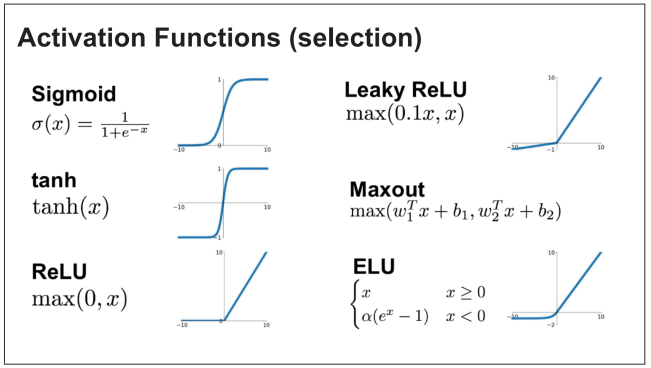
Here is another figure to visualize this relation (with slightly different notation)

{ *Credit*: Introduction to Neural Network Models of Cognition, by Pablo Caceres }


$\lfloor a_3 \rfloor$
x_1
x_2
W_{11}
W_{12}
W_{13}
W_{21}
W_{22}
W_{23}
z_1 $a_1 = \sigma(z_1)$
z_2 $a_2 = \sigma(z_2)$
z_3 $a_3 = \sigma(z_3)$
W_1
W_2
W_{31}
sigmoid activation
z_1 $a_1 = \sigma(z_1)$
decision threshold
$\hat{y} = f(a_m) \begin{cases} +1, & \text{if } a > 0.5 \\ -1, & \text{otherwise} \end{cases}$
input layer
hidden layer
output layer
classification

Activation Function

- The nonlinearity in the activation function $\sigma(\cdot)$ is essential because without it the prediction model would collapse into a linear model.
- What is a good choice for the activation function?



Activation Functions (selection)

Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$

tanh
$\tanh(x)$

ReLU
$\max(0, x)$

Leaky ReLU
$\max(0.1x, x)$

Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

{ *Credit*: Dani Bauer Lecture Notes }

- Early on, the sigmoid function was used.
- The preferred choice in modern neural networks is the ReLU (rectifed linear ReLU unit) activation function. It can be computed and stored more efficiently than a sigmoid activation.

Motivation - Universal approximation theorems

- These results state that a feedforward network with at least one hidden layer can approximate any function from one finite-dimensional space to another with any desired accuracy
    - Activation function must be nonlinear, such as the sigmoid activation function
    - The network must contain a sufficient number of hidden units.
- These theorems suggest that regardless of what function we are trying to learn, we know that a large NN will be able to represent this function.
    - We are not guaranteed that the training *algorithm* will be able to learn that function.
- I like to think of a Taylor-series expansion that allows one to approximate any function, using polynomials as the basis.

Some Caveats

- A feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly.
- In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.

Fitting the model

- Fitting a neural network requires estimating the unknown parameters.
  - For a quantitative response, typically squared-error loss is used.
- To overcome some of these issues and to protect from overfitting, two general strategies are employed when fitting neural networks:
  - *Slow Learning*: the model is fit in a somewhat slow iterative fashion, using gradient descent.
    - The fitting process is then stopped when gradient descent overfitting is detected.
  - *Regularization*: penalties are imposed on the parameters, usually lasso or ridge.

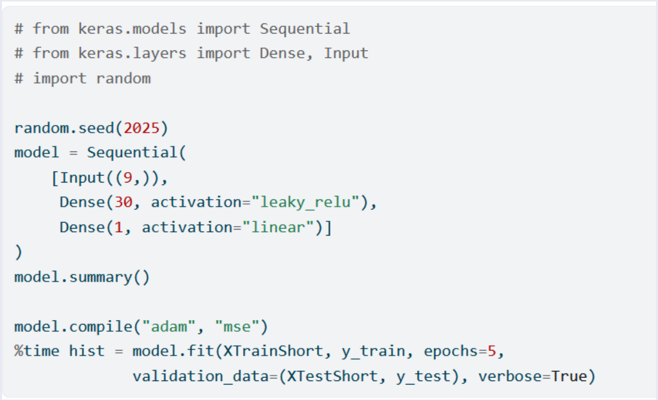
Data Splitting

- With deep learning problems, we typically enjoy large datasets.
    - When data are limited, use **cross-validation** (more common in classical ML).
- Split the data into training, validation, and test samples.
    - The validation set is used to:
        - Evaluate model performance *during training*
        - Tune hyperparameters *without touching the test set*
        - Prevents *overfitting* by monitoring generalization performance

1. *Split the data*: Train / Validation / (Test)
    - E.g., 80% train, 10% validation, 10% test
2. *Train multiple models* using different combinations of hyperparameters.
3. *Evaluate each model* on the validation set.
4. *Select the hyperparameter combination* that gives the best validation performance (e.g., highest accuracy, lowest loss).
5. Optionally, *retrain* the final model on train + validation and test it on the test set.

Seattle House Price Example

Here is some illustrative Python code to fit a neural net regression model.



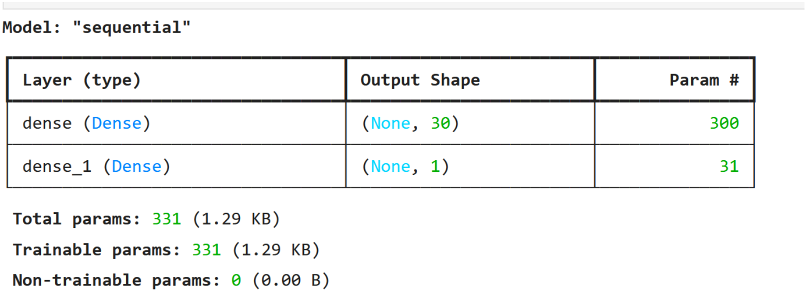
```python
# from keras.models import Sequential
# from keras.layers import Dense, Input
# import random

random.seed(2025)
model = Sequential(
    [Input((9,)),
     Dense(30, activation="leaky_relu"),
     Dense(1, activation="linear")]
)
model.summary()

model.compile("adam", "mse")
%time hist = model.fit(XTrainShort, y_train, epochs=5,
             validation_data=(XTestShort, y_test), verbose=True)
```

- In the first part (commented out here), various packages are imported (like the `library` function in `R`)
- A seed is set for the random number generator, to replicate results
- A *sequential* model is built, where layers are stacked linearly.
  - `Input((9,)):` Specifies the input has 9 features.
  - `Dense(30, activation="leaky_relu"):`
    - A hidden layer with 30 units using Leaky ReLU activation (good for "avoiding dead neurons").
  - `Dense(1, activation="linear"):` Typically linear for regression.

- Here is the result of `model.summary()`



```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ dense (Dense)                    │ (None, 30)             │           300 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                  │ (None, 1)              │            31 │
└──────────────────────────────────┴────────────────────────┴───────────────┘

 Total params: 331 (1.29 KB)

 Trainable params: 331 (1.29 KB)

 Non-trainable params: 0 (0.00 B)
```

- For the first dense (hidden) layer, there 9 + 1 (for the bias) parameters per neuron, so 300 in total.
- For the second dense (output) layer, there are weights for the 30 inputs plus 1 for the bias term, so 31 in total.

```
model.compile("adam", "mse")
%time
hist = model.fit(XTrainShort, y_train, epochs=5,
                 validation_data=(XTestShort, y_test),
                 verbose=True)
```

- In the `model.compile("adam", "mse")`, the optimizing algorithm is adam, an industry standard. The loss function is mean squared error, common in regression applications.
- The final statement:
    - Trains the model for 5 epochs using training data `(XTrainShort, y_train)`
    - Validates on `XTestShort, y_test` during training
    - `%time` shows how long the training takes
    - `hist` is an object that records training/validation loss over epochs

Gradient Descent

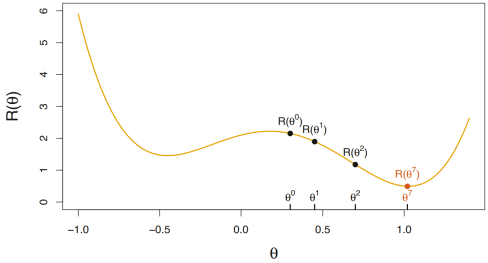
- **Gradient descent** is an iterative optimization algorithm used to **minimize a function** — most commonly a **loss function** in machine learning.
- You can think of it as a method to **descend a hill** (or surface) to find the lowest point — the **minimum** of a function.
- Suppose you have a function $L(\theta)$, where:
  - θ represents the model parameters
  - $L(\theta)$ is the **loss** (how bad your prediction is)

- At each step, you update the parameters to move **against the gradient** of the loss function:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta L(\theta_t)$$

Where:

  - η is the **learning rate** (a small positive number)
  - $\nabla_\theta L(\theta_t)$ is the **gradient** — a vector of partial derivatives that points in the direction of greatest increase
- You subtract the gradient to go in the direction of **steepest descent**.



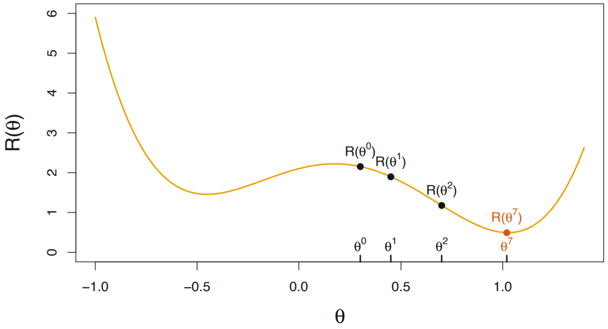


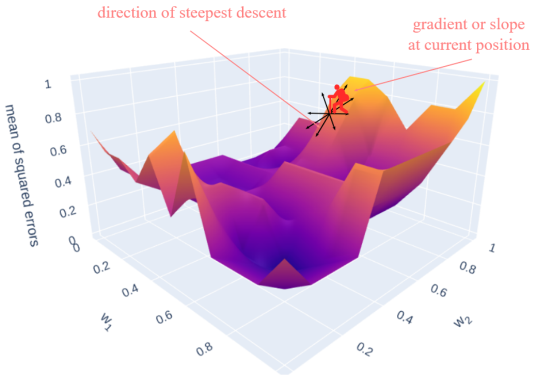
FIGURE 10.17. *Illustration of gradient descent for one-dimensional θ. The objective function $R(\theta)$ is not convex, and has two minima, one at $\theta = -0.46$ (local), the other at $\theta = 1.02$ (global). Starting at some value θ^0 (typically randomly chosen), each step in θ moves downhill — against the gradient — until it cannot go down any further. Here gradient descent reached the global minimum in 7 steps.*

{ *Credit*: James et al. (2023), An Introduction to Statistical Learning with Applications in Python }

- How to choose the learning rate?
  - Too small $\rightarrow$ slow convergence
  - Too large $\rightarrow$ overshooting, divergence
  - With the Adam optimizer, one utilizes adaptive learning rates

Visualization. Imagine a 3D surface of the loss function. At each step, gradient descent moves you "downhill" in the direction where the slo

Figure 5



{ *Credit*: Introduction to Neural Network Models of Cognition, by Pablo Caceres }

Stochastic Gradient Descent

- Nearly all of deep learning is powered by one very important variation: **stochastic gradient descent** (SGD).
- **The insight of SGD is that the gradient is an expectation.**
  - The expectation may be approximately estimated using a small set of samples.
  - Specifically, on each step of the algorithm, we can sample a minibatch of examples
- Instead of determining the gradient in the gradient decent step based on the full training data, we only use a subsample (a so-called **batch**) for estimating the gradient.
  - This subsample changes in every step or iteration (similar to K-fold cross-validation).
  - We cycle through the full data, so we are using a different batch in subsequent steps.

- The number of iterations to go through the full dataset is called an **epoch**.
- Using SGD can make it less likely to get stuck in local minima or saddle points, compared to full-batch gradient descent.
  - This is because **SGD adds noise**
- Each update in SGD is based on **a single (or small batch of) data point(s)**.
- The gradients are noisy approximations of the true gradient.
- This randomness helps the algorithm:
  - Escape shallow local minima
  - Jump over saddle points (flat regions with zero gradient but not minimums)
- That is why we often use mini-batch SGD (e.g., 32 or 64 samples per batch) — a sweet spot between stability and noise.

Want to try programming this from scratch? Check out:

- Introduction to Neural Network Models of Cognition (Section 6), by Pablo Caceres
- Dive Into Deep Learning, Section 5.2

Feedforward Multiple Layer Artificial Neural Network

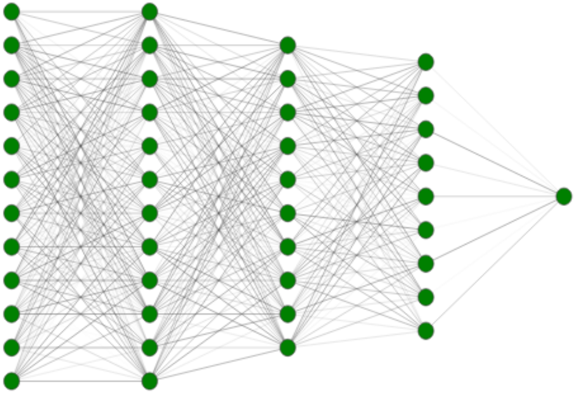
- They are called "feedforward" to emphasize the fact that information flows from the inputs, through the intermediate layers, to the outputs.
    - In contrast, later we will introduce "recurrent neural networks" in which outputs of the model are fed back into itself.
    - These models will be useful for time series data.
- When the number of layers, called the depth of the model, is large enough, the network is referred to as **deep**, leading to the so-called **deep neural network**.
- Writing equations is not hard but does require a bit of notation.
    - The idea is that the prediction function can be written as a connected chain of functions (i.e., composition of functions), of the form
    $$f = f^{(K)} \circ f^{(K-1)} \circ \circ \circ f^{(1)}$$
    where $\circ$ denotes *function composition.*

- Each hidden layer of the network is typically vector valued, with their dimensionality determining the width of the model.
  - In particular, each layer produces an intermediate vector $\mathbf{h}^{(i)}$ from the previous vector $\mathbf{h}^{(i-1)}$ of the form

$$\mathbf{h}^{(i)} = \mathbf{f}^{(i)}(\mathbf{h}^{(i-1)}) = \sigma^{(i)}(\mathbf{W}^{(i)}\ \mathbf{h}^{(i-1)})$$

- Note that the activation function $\sigma(\cdot)$ varies by layer i.

- For interpretation purposes, it is typically easier to present a graphical expression, such as:



Network Tuning

This network is considered to be relatively straightforward; it nevertheless requires a number of choices that all have an effect on the performance:

- *The number of hidden layers, and the number of units per layer.*
  - Modern thinking is that the number of units per hidden layer can be large, and overfitting can be controlled via the various forms of regularization.
- *Regularization tuning parameters*
  - These include the dropout rate and the strength of lasso and ridge regularization, and
  - are typically set separately at each layer.
- *Details of stochastic gradient descent*. These include
  - the batch size,
  - the number of epochs, and if used,
  - details of data augmentation (for convolutional networks).

Network Tuning 2

- The flexibility of neural networks is also one of their main drawbacks: there are **many** hyperparameters to tweak.
    - Not only can you use any imaginable network architecture, but even in a basic NN you can change:
        - the number of layers,
        - the number of neurons and the type of activation function to use in each layer,
        - the weight initialization logic,
        - the type of optimizer to use,
        - its learning rate,
        - the batch size,
        - and more.
- How do you know what combination of hyperparameters is the best for your task?

Backpropogation

- Backpropagation or **backprop** is an algorithm to conveniently calculate the prediction for a given feature vector and a given set of parameters as well as the corresponding gradient.
- Essentially the idea is to do
  - a sweep that goes forward through the network to determine the prediction (saving intermediary information) and then
  - going *backward* through the network and determining each of the partial derivatives to the parameters that make up the gradient.
- So, backprop is simply a clever way to determine all the partial derivatives of the prediction.

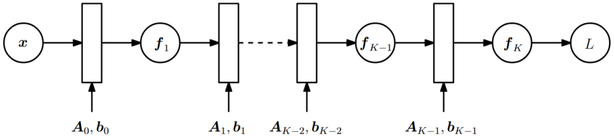
Backpropogation - Mathematics

Suppose we have outputs $\mathbf{y}$ that depend on inputs $\mathbf{x}$

$$\mathbf{y} = (f_K \circ f_{K-1} \circ \cdots \circ f_1(\mathbf{x}) = f_K(f_{K-1}(\cdots(f_1(\mathbf{x}))\cdots))$$

where each function f_i, $i = 1, \ldots, K$ possess its own parameters. In a neural net, we have $f_i(\mathbf{x}_{i-1}) = \sigma(\mathbf{A}_{i-1}\mathbf{x}_{i-1} + b_{i-1})$.

Figure 5.8 Forward pass in a multi-layer neural network to compute the loss L as a function of the inputs $\boldsymbol{x}$ and the parameters $\boldsymbol{A}_i$, $\boldsymbol{b}_i$.



$\boldsymbol{x}$
f_1
f_{K-1}
f_K
L
$\boldsymbol{A}_0, \boldsymbol{b}_0$
$\boldsymbol{A}_1, \boldsymbol{b}_1$
$\boldsymbol{A}_{K-2}, \boldsymbol{b}_{K-2}$
$\boldsymbol{A}_{K-1}, \boldsymbol{b}_{K-1}$

Think of the squared loss function

$$L(\theta) = ||\mathbf{y} - f_K(\theta, \mathbf{x})||^2$$

where $\theta = \{\mathbf{A}_0, b_0, \ldots, \mathbf{A}_{K-1}, b_{K-1}\}$.

- Use the chain rule to determine

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{K-1}} = \frac{\partial L}{\partial \boldsymbol{f}_K} \frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{\theta}_{K-1}}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{K-2}} = \frac{\partial L}{\partial \boldsymbol{f}_K} \boxed{\frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{f}_{K-1}} \frac{\partial \boldsymbol{f}_{K-1}}{\partial \boldsymbol{\theta}_{K-2}}}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{K-3}} = \frac{\partial L}{\partial \boldsymbol{f}_K} \frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{f}_{K-1}} \boxed{\frac{\partial \boldsymbol{f}_{K-1}}{\partial \boldsymbol{f}_{K-2}} \frac{\partial \boldsymbol{f}_{K-2}}{\partial \boldsymbol{\theta}_{K-3}}}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{i}} = \frac{\partial L}{\partial \boldsymbol{f}_K} \frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{f}_{K-1}} \cdots \boxed{\frac{\partial \boldsymbol{f}_{i+2}}{\partial \boldsymbol{f}_{i+1}} \frac{\partial \boldsymbol{f}_{i+1}}{\partial \boldsymbol{\theta}_{i}}}$$

- The orange terms are partial derivatives of the output of a layer with respect to its inputs
- The blue terms are partial derivatives of the output of a layer with respect to its parameters.
- Assuming we have already computed the partial derivatives $\partial L/\partial\theta_{i+1}$, then most of the computation can be reused to compute $\partial L/\partial\theta_i$. **Backwards!**

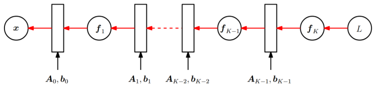
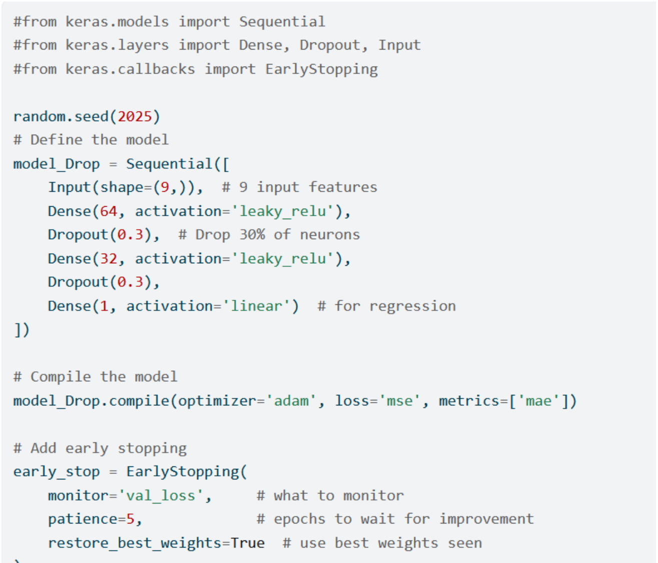

$\boldsymbol{x}$
$\boldsymbol{f}_1$
$\boldsymbol{f}_{K-1}$
$\boldsymbol{f}_K$
L
$\boldsymbol{A}_0, \boldsymbol{b}_0$
$\boldsymbol{A}_1, \boldsymbol{b}_1$
$\boldsymbol{A}_{K-2}, \boldsymbol{b}_{K-2}$
$\boldsymbol{A}_{K-1}, \boldsymbol{b}_{K-1}$

Figure 5.9 Backward pass in a multi-layer neural network to compute the gradients of the loss function.

Early Stopping

- When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again.
    - This means we can obtain a model with better validation set error by returning to the parameter setting at the point in time with the lowest validation set error.
- Every time the error on the validation set improves, we store a copy of the model parameters.
    - When the training algorithm terminates, we return these parameters, rather than the latest parameters.
    - The algorithm terminates when no parameters have improved over the best recorded validation error for some pre-specified number of iterations.
- This strategy is known as **early stopping**. It is probably the most commonly used form of regularization in deep learning. Its popularity is due to both its effectiveness and its simplicity.

Here is some illustrative Python code to illustrate early stopping and dropout, the next section.



```
#from keras.models import Sequential
#from keras.layers import Dense, Dropout, Input
#from keras.callbacks import EarlyStopping

random.seed(2025)
# Define the model
model_Drop = Sequential([
    Input(shape=(9,)),  # 9 input features
    Dense(64, activation='leaky_relu'),
    Dropout(0.3),  # Drop 30% of neurons
    Dense(32, activation='leaky_relu'),
    Dropout(0.3),
    Dense(1, activation='linear')  # for regression
])

# Compile the model
model_Drop.compile(optimizer='adam', loss='mse', metrics=['mae'])

# Add early stopping
early_stop = EarlyStopping(
    monitor='val_loss',      # what to monitor
    patience=5,              # epochs to wait for improvement
    restore_best_weights=True  # use best weights seen
```

```python
# Add early stopping
early_stop = EarlyStopping(
    monitor='val_loss',      # what to monitor
    patience=5,              # epochs to wait for improvement
    restore_best_weights=True  # use best weights seen
)

# Train the model
history_Drop = model_Drop.fit(
    XTrainShort_sc, y_train,
    validation_data=(XTestShort_sc, y_test),
    epochs=150,
    batch_size=32,
    callbacks=[early_stop],
    verbose=False
)
```

In the code for `EarlyStopping`

- `monitor='val_loss'` means stop based on validation loss
- `patience=5` means allow 5 epochs of no improvement
- `restore_best_weights=True` means revert to the best model

Dropout Learning

- Dropout is one of the most effective and most commonly used regularization techniques for neural networks;
- Inspired by random forests, the idea is to randomly remove a fraction of the units in a layer when fitting the model.
- Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training.

- The core idea is that introducing noise in the output values of a layer can break up happenstance patterns that aren't significant, which the model will start memorizing if no noise is present.
- As your model starts overfitting, your goal switches to improving generalization through model regularization. You can reduce your model's capacity, add dropout or weight regularization, and use early stopping. And naturally, a larger or better dataset is always the number one way to help a model generalize.
- In the code,
  - `Dropout(rate)`, rate $= 0.3$ means randomly turn off 30% of units during training.
  - Used only during training (automatically skipped during evaluation/prediction).

Session 4A. Artificial Neural Networks Summary

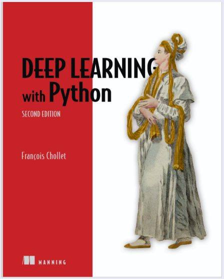
This module covered:

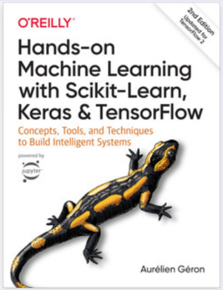
- a bit of neural net history
- the single layer feedforward network, including the model definition and approximation theorems
- fitting the model, illustrated by the Seattle House Price example
- the multiple layers feedforward network, including network tuning, backpropogation, and so on.

During lab, participants may follow the notebooks Seattle House Prices and Claim Counts.

Resources For Future Studies

- Chollet, F. (2021). Deep Learning with Python
- Géron, A. (2022). Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow (3rd ed.). O'Reilly Media.




O'REILLY
2nd Edition Updated for TensorFlow 2
Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow
Concepts, Tools, and Techniques to Build Intelligent Systems
powered by jupyter
Aurélien Géron