Neural network 2

Dr. Kelly Trinh

Housekeeping

- Assessment 1 due on Sunday (16/5)
  - change the setting from "Private" to "Viewable"

Housekeeping

- Assessment 1 due on Sunday (16/5)
  - change the setting from "Private" to "Viewable"
- Register AWS free tier account

Housekeeping

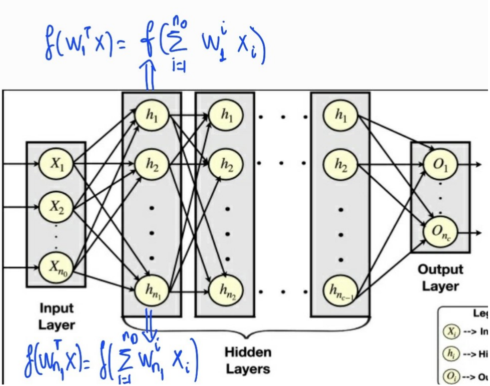
- Assessment 1 due on Sunday (16/5)
  - change the setting from "Private" to "Viewable"
- Register AWS free tier account
- Be familiar with Python and know how to implement a NN using Tensorflow

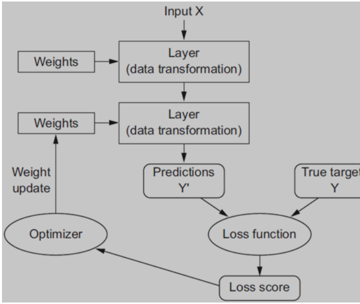
Housekeeping

- Assessment 1 due on Sunday (16/5)
  - change the setting from "Private" to "Viewable"
- Register AWS free tier account
- Be familiar with Python and know how to implement a NN using Tensorflow

Outline

- Optimisation
- Regularisation
- Vanishing/Exploding Gradient Problems
- Week 4 and 5 Online materials


$f(w_1^T x) = f\left(\sum_{i=1}^{n_0} w_1^i x_i\right)$
$f(w_{n_1}^T x) = f\left(\sum_{i=1}^{n_0} w_{n_1}^i x_i\right)$
Input Layer
Hidden Layers
Output Layer


Input X
Weights
Layer (data transformation)
Weights
Layer (data transformation)
Weight update
Predictions Y'
True target Y
Optimizer
Loss function
Loss score

Recall from the last week-Gradient Descent

1. Initialize weights randomly
2. Repeat until convergence $\{\ \mathbf{W}_k \leftarrow \mathbf{W}_{k-1} - \alpha \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}_{k-1}}\ \}$
3. Return weights

1. Initialize weights randomly
2. Repeat until convergence $\{ \mathbf{W}_k \leftarrow \mathbf{W}_{k-1} - \alpha \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}_{k-1}} \}$
3. Return weights

where

- α: learning rate $\rightarrow$ How to choose the learning rate?

Recall from the last week-Gradient Descent

1. Initialize weights randomly
2. Repeat until convergence $\{ \mathbf{W}_k \leftarrow \mathbf{W}_{k-1} - \alpha \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}_{k-1}} \}$
3. Return weights

where

- α: learning rate $\rightarrow$ How to choose the learning rate?
- $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}_{k-1}}$ tell us how much the loss $J(\mathbf{W})$ change due to a small change in $\mathbf{W}$
- **Problems**:
  - Gradient computation can be demanding when data is large, particularly in image detection

Recall from the last week-Gradient Descent

1. Initialize weights randomly
2. Repeat until convergence $\{ \mathbf{W}_k \leftarrow \mathbf{W}_{k-1} - \alpha \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}_{k-1}} \}$
3. Return weights

where

- α: learning rate $\rightarrow$ How to choose the learning rate?
- $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}_{k-1}}$ tell us how much the loss $J(\mathbf{W})$ change due to a small change in $\mathbf{W}$
- **Problems**:
  - Gradient computation can be demanding when data is large, particularly in image detection
  - Might reach to different local optimal depending on starting points

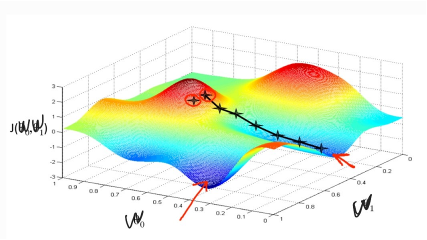
Recall from the last week-Gradient Descent

1. Initialize weights randomly
2. Repeat until convergence $\{\ \mathbf{W}_k \leftarrow \mathbf{W}_{k-1} - \alpha \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}_{k-1}}\ \}$
3. Return weights

where

- α: learning rate $\rightarrow$ How to choose the learning rate?
- $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}_{k-1}}$ tell us how much the loss $J(\mathbf{W})$ change due to a small change in $\mathbf{W}$
- **Problems**:
  - Gradient computation can be demanding when data is large, particularly in image detection
  - Might reach to different local optimal depending on starting points
  - Finding the global optimum is very hard and can get stuck in the local optimum

.



Source: TowardsDataScience

How to select **learning rate**, α?

- **Learning schedules**
  - Power scheduling: learning rate is a function of the iteration number n: $\alpha_t = \alpha_0/(1 + n/s)^c$ where α_0 is initial learning rate, c is often set to 1, s is a hyperparameter

How to select **learning rate**, α?

- **Learning schedules**
  - Power scheduling: learning rate is a function of the iteration number n: $\alpha_t = \alpha_0/(1 + n/s)^c$ where α_0 is initial learning rate, c is often set to 1, s is a hyperparameter
  - After s steps, the learning rate is reduced to $\alpha_0/2$, and after another s steps, the learning rate is reduced to $\alpha_0/3$ and so on.

```
# Power scheduling (decay is equivalent to 1/s)
opt = tf.keras.optimizers.SGD(lr=0.1, decay=1e-3)
model.compile(optimizer=opt, loss ='mse')
```

Learning rate

- **Learning schedules**
  - Exponential scheduling $\alpha(t) = \alpha_0 0.1^{n/s}$. The learning rate will drop by a factor of 10 every s steps.

```
# Exponential scheduling

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

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
# And then create a LearningRateScheduler callback and pass this call back to fit(
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(exponential_decay_fn)
model.compile(tf.keras.optimizers.SGD(), loss='mse')
model.fit(X_train, Y_train, callbacks=[lr_scheduler])
```

- Learning Schedules
  - Piecewise constant scheduling: use a constant learning rate for a number of epoches, and then a smaller learning rate for another epoches.

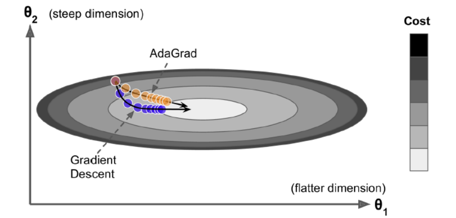
`patience`: number of epochs with no further improvement after which training will be stopped.

```
# Piecewise constant

def piecewise_fn(epoch):
    if epoch < 10:
        return 0.01
    elif epoch < 30:
        return 0.005
    else:
        return 0.001
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(piecewise_fn)
model.fit(X_train, Y_train, callbacks=[lr_scheduler])

# Or can use tf.keras.callbacks.ReduceROnPlateau
lr_scheduler = tf.keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=15)

#The learning rate will be multiplied by 0.5 when the best validation loss does not imp
epochs
```

- **Adaptive learning rates**: Adagrad, RMSProp, Adadelta, Adam. The learning rate depends on magnitude of gradient, size of particular weights.

Gradient descent (GD)-Stochastic GD-Mini Batch GD

- **Gradient Descent**: The gradient $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ is computed using the entire data. Sometimes it is challenging to compute the gradient.
- **Stochastic gradient descent**: compute the gradient at single point data; $\frac{\partial J(\mathbf{W}; X^{(j)}, Y^{(j)})}{\partial \mathbf{W}_{j-1}}$ However, the estimation can be noisy
- **Min-Batch gradient descent**: the gradient is computed using a subset of data; $\frac{\partial J(\mathbf{W}; X^{(j:j+m)}, Y^{(j:j+m)})}{\partial \mathbf{W}}$. Smoother convergence and can be used in parallelization .

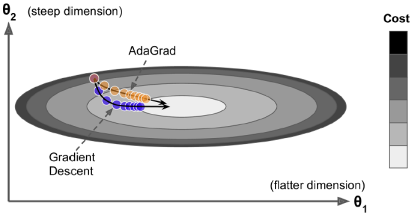


- **Adagrad**: adapts the learning rate to the parameters associated with steepest slope. The correction is done by scaling down the gradient vector by the amount $\sqrt{s+\epsilon}$ where ϵ, smoothing term, is often set at low value such as 10^{-6}.

$$s \leftarrow s + \nabla_{\mathbf{W}} J(\mathbf{W}) \otimes \nabla_{\mathbf{W}} J(\mathbf{W})$$

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha_t \nabla_{\mathbf{W}} J(\mathbf{W}) \oslash \sqrt{s+\epsilon}$$

Other optimizers



- **Adagrad**: adapts the learning rate to the parameters associated with steepest slope. The correction is done by scaling down the gradient vector by the amount $\sqrt{s+\epsilon}$ where ϵ, smoothing term, is often set at low value such as 10^{-6}.

$$s \leftarrow s + \nabla_{\mathbf{W}} J(\mathbf{W}) \otimes \nabla_{\mathbf{W}} J(\mathbf{W})$$

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha_t \nabla_{\mathbf{W}} J(\mathbf{W}) \oslash \sqrt{s+\epsilon}$$

- It does not perform well for deep neural network.
- Tensorflow syntax: `tf.keras.optimizers.Adagrad`

```
tf.keras.optimizers.Adagrad(
    learning_rate=0.01, initial_accumulator_value=0.1, epsilon=1e-07)
```

Other optimizers

- **RMSProp**: only accumulate the gradients from the most recent iterations.

$$s \leftarrow \rho s + (1-\rho)\nabla_{\mathbf{W}} J(\mathbf{W}) \otimes \nabla_{\mathbf{W}} J(\mathbf{W})$$
$$\mathbf{W} \leftarrow \mathbf{W} - \alpha_t \nabla_{\mathbf{W}} J(\mathbf{W}) \oslash \sqrt{s+\epsilon}$$

ρ is often set at 0.9.

- Tensorflow syntax: `tf.keras.optimizers.RMSProp`

```
tf.keras.optimizers.RMSprop(
    learning_rate=0.001, rho=0.9, epsilon=1e-07)
```

Other optimizers

- **Momentum**: consider the magnitude of previous gradient descent, therefore it helps to accelerate gradient descent to relevant direction

$$m_t = \beta m_{t-1} - \alpha \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$$

$$\mathbf{W}_t = \mathbf{W}_{t-1} + m_t$$

```
# Momentum

tf.keras.optimizers.SGD(
    learning_rate=0.01, momentum=0.9, nesterov=False, name='SGD')
```

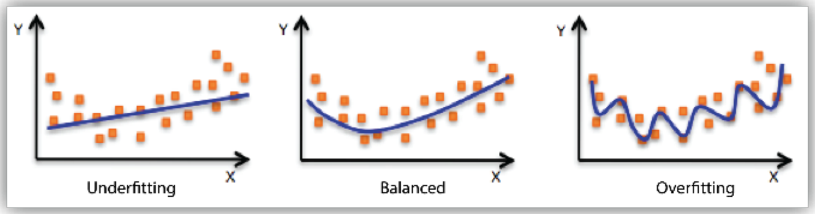
Optimizers

- **Adam**:(adaptive moment estimation) combines the ideas of momentum optimisation and RMSpropt.
- It is a common algorithm used in deep neural network.

Over-fitting issue

- Regularisation
- Dropout
- Early stop

Over-fitting



- Under-fitting: a model cannot fully learn the data
- Over-fitting: a model are to complex and often result in poor prediction and classification generalise the model

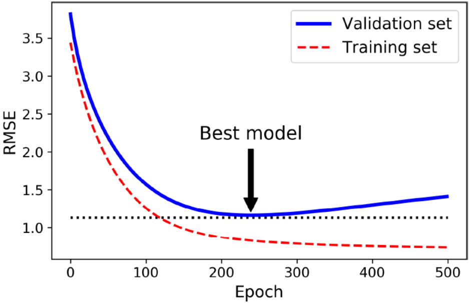
L^1, L^2 Regularisation

- Adding a penalty to the loss function in proportion to the size of the weights
$$L^*(\mathbf{W}) = L(\mathbf{W}) + \alpha\Omega(\mathbf{W})$$
where α controls the amount of shrinking; $\Omega(.)$ is the penalisation function.
- L^2 regularisation also known as a weight decay, ridge regression. $\Omega(\mathbf{W}) = \frac{1}{2}\sum_{i=1}^{n} w_i^2$
- L^1 regularisation $\Omega(\mathbf{W}) = \frac{1}{2}\sum_{i=1}^{n} w_i$
- Python syntax: `tf.keras.regularizers.l1`, `tf.keras.regularizers.l2`

```
from functools import partial
RegularizedDense = partial(tf.keras.layers.Dense,
activation="relu",
kernel_regularizer=tf.keras.regularizers.l2(0.01))


model = tf.keras.Sequential([
tf.keras.layers.Flatten(input_shape=[28, 28]),
RegularizedDense(300),
RegularizedDense(100),
RegularizedDense(10, activation="softmax")
])
```

Early Stopping Regularization

- The model stops training as soon as the validation error reach to a minimum
- Python syntax `tf.keras.callbacks.EarlyStopping`

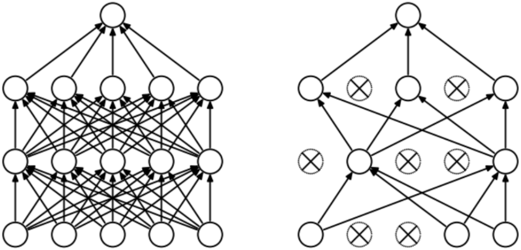


Early Stopping Regularization

```python
#Early Stopping

callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=15)

model.fit(trainX, trainY, validation_data=(testX, testy),
          epochs=4000, callbacks=[callback])
```

Drop-out

- Randomly set some nodes to 0. The *dropout rate* is often set between 10% and 50%.
- Python syntax: `tf.keras.layers.Dropout`



(a) Standard Neural Net (b) After applying dropout.

Source: Medium

```
# Dropout 20%

tf.keras.layers.Dropout(.2)
```

Vanishing/Exploding Gradient Problems

Vanishing/Exploding Gradient Problems

- **Vanishing gradient problem:** Gradients become smaller and smaller when the optimiser reach to the lower layers.
- **Exploding gradient problem:** Gradient become larger and lager for lower layers. This often happens in recurrent neural network
- 3 popular ways to handle the unstable gradient problems are
  - Initialization
  - Activation function
  - Batch Normalization

Initialization

- Glorot and Bengio proposed a initialization for weights and bias to alleviate the unstable gradient problem.
- The *Glorot initialization* using for the logistic activation function
  - $\mathcal{N}(0, \frac{1}{fan_{avg}})$
  - $\mathcal{U}(-\sqrt{\frac{3}{fan_{avg}}}, \sqrt{\frac{3}{fan_{avg}}})$
  - where $fan_{avg} = (fan_{in} + fan_{out})/2$, fan_{in} is the number of inputs to a hidden unit and fan_{out} is the number of outputs to a hidden unit

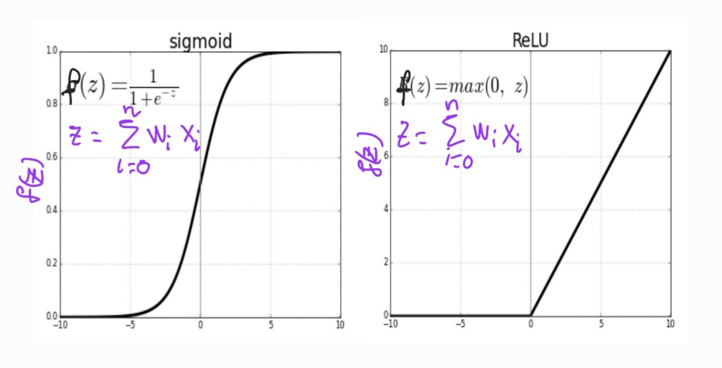
```
tf.keras.layers.Dense(10, activation="tanh",  kernel_initializer='glorot_uniform')
```

Initaliztation

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, Tanh, Logistic, Softmax	$1 / fan_{avg}$
He	ReLU & variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

Source: Aurelien Geron (2019)
Python syntax: `he_uniform`, `he_normal`

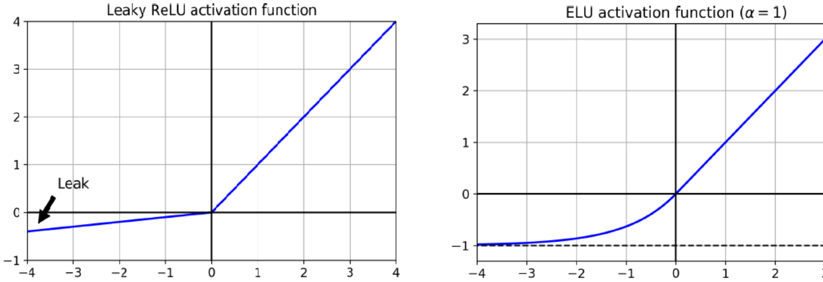
Activation functions



Source: Towards Data Science

- Sigmoid: vanishing gradient
- ReLu: deying ReLu problem- some neurons "die", only produce the output of zeros.

Activation function



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

- ELU (exponential linear unit) has a slower computation than ReLu and Leaky ReLu

Batch Normalisation

- The initialization approach and activation function do not guarantee that the unstable gradient will not happen during training.
- Ioffe and Szegedy proposed a technique- *Batch Normalization*- to address the unstable gradient during the training of deep neural networks.
- Batch Normalization applies zero-center and normalizes each input, then scales and shifts the results using 2 new parameters per layers (scaling and shifting parameters). The scaling and shifting parameters are determined by the algorithm.

Batch Normalisation


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

Homework

- Implement the techniques learnt today on the code provided in Practical Session in Week 1 to investigate their performance on NN.

Lab session

- Introduction to AWS-SageMaker
- Implementation of NN in AWS-SageMaker

- NO TECHNICAL SESSION ON WEDNESDAY, WEEK 3
- WEEK 4: CNN

Further references

- "Hands-on MAchine Learning with Scikit-Learn Keras and Tensorflow" Aurelien Geron (2019)
- "Improving neural networks by preventing co-adaptation of feature detectors" G. Hinton et al. (2012).
- "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," N. Srivastava et al. (2014).
- "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," S. Ioffe and C. Szegedy (2015)